© 2018 Johnathan Alsop

SPECIALIZATION WITHOUT COMPLEXITY IN HETEROGENEOUS MEMORY SYSTEMS

BY

JOHNATHAN ALSOP

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Chair & Director of Research
Professor Deming Chen
Associate Professor Rakesh Kumar
Professor Marc Snir
Dr. Bradford Beckmann, AMD Research
Professor David Wood, University of Wisconsin - Madison

Abstract

The end of Dennard scaling and Moore's law has motivated a rise in the use of parallelism and hardware specialization in computer system design. Across all compute domains, applications have increasingly relied on specialized devices such as GPUs, DSPs, FPGAs, etc., to execute tasks faster and more efficiently, but interfacing these diverse devices within a heterogeneous system remains an important challenge. Early heterogeneous systems were loosely coupled and lacked a shared coherent memory interface, so specialization was reserved for highly regular code patterns with coarse-grained synchronization requirements. More recently, the need to accelerate applications with more irregular and fine-grained sharing patterns has led to significant research into closer integration of specialized devices.

A single global address space enables improved programmability, communication efficiency, data reuse, and load balancing for emerging heterogeneous applications. Consequently, there have been many attempts to integrate specialized devices and their caches into a single coherent memory hierarchy to improve performance in future systems-on-chip (SoCs). However, coherence is particularly difficult to implement in heterogeneous systems. Differences in parallelism, locality, and synchronization in high-throughput accelerators such as GPUs means that coherence and consistency strategies designed for CPUs are ineffective, and evaluating the performance of alternative strategies is difficult. Recent efforts to implement coherence for such devices involve a simple software-driven coherence strategy combined with complex extensions to a conventional memory consistency model, which guarantees sequential consistency (SC) for programs that are data race-free (DRF). The first extension, scoped synchronization, avoids coherence costs when synchronization is guaranteed to be local, but it requires the use of the heterogeneous race-free (HRF) consistency model, which limits sharing patterns and increases the burden on the programmer. The second extension, relaxed atomics, allows the programmer to avoid costly ordering constraints when they are unnecessary

for functionality, but existing consistency models offer complex and often poorly specified semantics when relaxed atomics are used. Once an appropriate coherence and consistency strategy is determined for a device, interfacing it with devices using different strategies poses another critical challenge. Existing integration strategies are incremental, either sacrificing system flexibility or incurring significant added complexity to achieve this goal. A rethinking of heterogeneous coherence and protocol integration from the ground up is needed.

This work lays out a path to implementing flexible and efficient heterogeneous coherence without adding complexity to the consistency model or the system design. To help understand the memory demands of emerging specialized hardware, we first describe a performance analysis tool we developed for highly parallel workloads. Insights from this tool helped guide the development of a collection of coherence and consistency innovations for high-throughput accelerators. On the coherence side, we describe two innovations, DeNovo for GPUs and heterogeneous lazy release consistency (hLRC), which demonstrate that scoped synchronization is not necessary for cache efficiency in high-throughput devices. On the consistency side, this work describes the DRFrlx consistency model, which formalizes safe use cases of atomic relaxation. Again, we offer these benefits while retaining a simple SC-centric DRF consistency model. Finally, to address the challenge of integrating diverse coherence strategies, we present the Spandex coherence interface. Spandex can flexibly and simply integrate devices with a broad range of memory demands in an SoC, and we show how this flexibility enables new performance optimizations that can take advantage of hints about the expected memory demands of an application. Together, these innovations establish a framework for integrating future SoCs that can dynamically adapt to serve the diverse memory demands of future accelerators without incurring complexity for hardware or software designers.

To Rory.

Acknowledgments

I am fortunate to count many people who have supported me in my academic career. First I'd like to thank all those who served on my thesis committee - Sarita Adve, Brad Beckmann, Deming Chen, Nam Sung Kim, Rakesh Kumar, Marc Snir, and David Wood - who provided valuable feedback and guidance to the development and refinement of this work. Special thanks to Rakesh Kumar, who graciously joined close to the last minute, helping me to avert a minor crisis.

Thank you to all my coauthors on this work: Matt Sinclair, Rakesh Komuravelli, Marc Orr, Brad Beckmann, David Wood, and Sarita Adve. These projects were a collaborative effort, and the research would not have been possible, much less as successful or enjoyable as it was, without each of their contributions. I would gladly collaborate with any of these authors again in the future.

I'd like to thank my advisor, Professor Sarita Adve, for her dedicated guidance, motivation, and support. I am extremely lucky and honored to have had her as an advisor and role model. Her ability to think long term with a focus on the big picture has been critical to the conception, refinement, and completion of most of the innovations in this document. Her persistence and commitment to the students and good ideas that come out of her group have kept our projects moving forward through the low points, and (along with her presentation abilities) have brought attention to our successes. I hope some of this skill and passion has rubbed off on me over the years, and I look forward to working with her in the future.

I'd like to thank the other students and undergraduate researchers in my group - Matt Sinclair, Rakesh Komuravelli, Hyojin Sung, Radha Venkatagiri, Abdulrahman Mahmoud, Muhammad Huzaifa, Gio Salvador, Adel Ejjeh, Khalique Ahmed, Lin Cheng, and Weon Taek Na - for their help and friendship over the years. I especially want to thank Matt Sinclair for his mentorship. As the lead author on much of the work in this document, Matt was a major force in shaping many of these projects and guiding them to completion, and I am very lucky to have worked with him for so long

on so many successful projects. I couldn't have asked for a more selfless and capable mentor in my graduate career. Special thanks also to Hyojin Sung and Rakesh Komuravelli for their support in my first years at a time when I required frequent help navigating a new research environment.

Thank you to Brad Beckmann, another important mentor during and after my time as an intern at AMD Research. Without his guidance, commitment, and insights the hLRC work could not have happened. Additionally, his example in navigating the world of academia from a position in industry helped solidify my interest in industry research. I'm excited to be working again with him in my first job out of graduate school.

I'd like to thank all of my teachers and professors who have instructed and inspired me throughout my life, especially Mikko Lipasti at the University of Wisconsin. Mikko inspired my initial interest in computer systems as my professor in ECE 151, and in subsequent interactions he helped fuel that passion, as well as an interest in graduate study.

This work was supported in part by the National Science Foundation under grants CCF-1018796 and CCF-1302641, and by the Center for Future Architectures Research (C-FAR) and the Applications Driving Architectures (ADA) center, Semiconductor Research Corporation program sponsored by MARCO and DARPA. I have also received generous support from the Dan Vivoli Endowed Fellowship and the Rambus Computer Engineering Fellowship.

Thank you to my friends - including but not limited to Jamen, Jeremy, Geoff, Anne, Munoz, Graham, BK, Zia, Dan, Ellen, Sarah, Ta-Hsuan, Ivan, David, Katie, Burt, Heather, Anna, Gio, Kenny, and all the Quality regulars - for helping me stay balanced and being an incredible source of joy and fun times.

Thanks to my dogs, Maeby and Shirley, for mostly being good girls.

Thanks to my family for their constant love and encouragement, and for supporting me in everything I do. My parents, Bob and Karen, have been there for me since day one doing everything everything in their power to ensure my happiness and success. My brother, Jim, has been a good friend starting not long after that. My grandparents, Mary, Chuck, Jean, and Don, have also been significant role models and sources of wisdom, support, and encouragement in my life. My cousins, aunts, uncles, in-laws, and nephew are some of the funniest and kindest people I know, and I've always looked forward to opportunities to spend time with them. Overall I count myself genuinely lucky to be related to such good and loving people whose company I so thoroughly enjoy, and whose

support I know I can count on whenever I need it.

Finally, thank you to my wife Rory for loving me, supporting me, and generally being my favorite person. She has contributed to my passion for learning and research through her dedication, her encouragement, and her own considerable achievements. At the same time she inspires an appreciation for life's other pleasures, whether that means relaxing at home with a movie or seeking out fun events and experiences down the street or across the world. My life would be infinitely less exciting and enjoyable without her in it, and I would be much less self-aware.

Table of Contents

List of Tables . . . . **xi**

List of Figures . . . . **xii**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 5
1.2.1 GPU Stall Inspector: Performance Characterization in High-Throughput Accelerators 5
1.2.2 Efficient GPU Coherence without Scopes 6
1.2.3 DRF-Relaxed: Atomic Relaxation with SC-Centric Semantics 6
1.2.4 Spandex: Efficiently Integrating Heterogeneous Coherence Strategies 7
1.2.5 Dynamic Coherence Specialization 7
1.3 Thesis Organization 7

Chapter 2 GPU Stall Inspector: Performance Characterization in High-Throughput Accelerators **9**
2.1 Stall Classification in GSI 10
2.1.1 Classifying Causes of Instruction Stalls 10
2.1.2 Attributing Stalls to Cycles 11
2.1.3 Sub-Classifying Memory Data Stalls 12
2.1.4 Sub-Classifying Memory Structural Stalls 12
2.2 Case Study: Stash vs. Scratchpad+DMA 13
2.2.1 Background 14
2.2.2 Methodology 16
2.2.3 Evaluation 17
2.2.4 MSHR Sensitivity 19
2.3 Summary 20

Chapter 3 Efficient GPU Coherence Without Scopes **22**
3.1 Background 22
3.1.1 Data Race Free Memory Models (GPU coh.+DRF) 23
3.1.2 Scoped Synchronization (GPU coh.+HRF) 23
3.1.3 Remote Scope Promotion (GPU coh.+HRF+RSP) 25
3.2 DeNovo Coherence for GPUs (DeNovo+DRF) 25
3.2.1 DeNovo Coherence for Multicore CPUs 25
3.2.2 Qualitative Insights 26
3.2.3 Evaluation 27

3.3 Heterogeneous Lazy Release Consistency (hLRC+DRF) 33
3.3.1 Design 34
3.3.2 Evaluation 36
3.4 Summary 42

Chapter 4 DRF-Relaxed: Relaxed Atomics with SC-Centric Semantics 45
4.1 Background: Relaxed Atomics 45
4.2 Relaxed Atomics Use Cases 46
4.2.1 Unpaired Atomics 47
4.2.2 Non-Ordering Atomics 49
4.2.3 Speculative Atomics 50
4.2.4 Quantum Atomics 51
4.3 DRF-Relaxed Formal Definitions 52
4.3.1 Definitions for an SC Execution with SC total order T 53
4.3.2 DRF-Relaxed Model Definition 55
4.4 Formalization in Herd 56
4.4.1 Relaxed Atomics Herd Formalization 56
4.5 Evaluation 58
4.6 Summary 60

Chapter 5 Spandex: Efficiently Integrating Heterogeneous Coherence Strategies 64
5.1 Background 64
5.1.1 Protocol Classification 65
5.1.2 Heterogeneous Coherence Solutions 68
5.2 Design 69
5.2.1 Device Request Interface 70
5.2.2 Last Level Cache (LLC) 73
5.2.3 External Request Interface 78
5.2.4 Translation Unit Responsibilities 80
5.2.5 Consistency Requirements 82
5.2.6 Overheads 83
5.3 Methodology 84
5.3.1 Cache Configurations 84
5.3.2 Benchmarks 85
5.4 Evaluation 88
5.4.1 Synthetic Microbenchmarks 89
5.4.2 Collaborative Applications 90
5.5 Summary 92

Chapter 6 Dynamic Coherence Specialization 96
6.1 Specialized Coherence Overview 96
6.2 Complexity Analysis 99
6.3 Evaluation 101
6.3.1 Methodology 101
6.3.2 Results 103
6.4 Summary 107

Chapter 7 Related Work 108
7.1 Performance Evaluation for High-Throughput Accelerators 108
7.2 Efficient Heterogeneous Coherence and Coherence Integration 109
7.3 Atomic Relaxation in Heterogeneous Systems 111
7.4 Dynamic Coherence Specialization 111
7.5 Summary 113

Chapter 8 Conclusion and Future Work 114
8.1 Summary of Thesis 114
8.2 Future Work and Impact 115
8.2.1 Near-Term Research 115
8.2.2 Long-Term Research 116
8.2.3 Future Impact 117

References 119

List of Tables

2.1 Parameters of the simulated heterogeneous system. 16

3.1 Tradeoffs of each GPU coherence and consistency strategy. 23
3.2 Benchmarks with input sizes. All thread blocks (TBs) in the synchronization microbenchmarks execute the critical section or barrier many times. Microbenchmarks with local and global scope are denoted with a '_L' and '_G', respectively. 32
3.3 Simulated heterogeneous system parameters. 32
3.4 Simulated heterogeneous system parameters. 36
3.5 Workloads and inputs. 36

5.1 Coherence strategy classification. 65
5.2 Type and granularity of requests generated for read misses, write misses, and replacements of owned data in GPU coherence, DeNovo, and MESI caches. *A DeNovo ReqV request is issued at word granularity, but the responding device may include any available up-to-date data in the line. 72
5.3 The state transition triggered at the LLC by each request type (Next State) and the request type forwarded to the owning core in the event the data is in O state (Fwd Msg). An entry of – indicates no transition or forwarded request is necessary. 75
5.4 The state transition and response message triggered at a device by each external Spandex request. If the target data is not in the expected state when a request arrives, different behavior may be required (discussed in Section 5.2.3). 78
5.5 Simulated cache configurations. 85
5.6 Simulated heterogeneous system parameters. 86
5.7 Collaborative applications communication patterns and execution parameters. CTs = CPU threads. TBs = GPU thread blocks. 88

List of Figures

1.1 Coherence interface in a future heterogeneous system. 3

2.1 Stall cycle breakdowns for implicit microbenchmark (normalized to baseline scratchpad). 17
2.2 Stall cycle breakdowns for implicit microbenchmark with varying MSHR sizes (normalized to baseline scratchpad with 32-entry MSHR). 19

3.1 GPU coherence (*G**) and DeNovo (*D**) configurations with real GPU applications (no intra-kernel synchronization), normalized to *G**. 29
3.2 GPU coherence (*G**) and DeNovo (*D**) configurations with synchronization benchmarks that use only global synchronization, normalized to *G**. 30
3.3 All configurations with synchronization benchmarks that use mostly local synchronization, normalized to *GD*. 31
3.4 Performance (speedup relative to baseline). 37
3.5 Breakdowns of invalidation and flush action types. 38
3.6 Total latency breakdowns for data accesses and synchronization actions. 39

4.1 Relaxed atomics speedup on a discrete GPU. 46
4.2 Execution time and energy for microbenchmarks, normalized to *GD0*. 61
4.3 Execution time and dynamic energy for real applications, normalized to *GD0*. 62

5.1 Comparison of Spandex and a common existing integration strategy. 65
5.2 Overview of Spandex design. Requests to and from the LLC travel through a generalized network, which may be unordered. 70
5.3 Handling basic request types at the Spandex LLC. 77
5.4 Synthetic microbenchmarks execution time and network traffic, normalized to HMG. 94
5.5 Collaborative applications execution time and network traffic, normalized to HMG. . 95

6.1 State space comparison between Spandex and ARM CHI with added optimizations, writethrough forwarding (WTfwd) and owner prediction (Opred). 100
6.2 Synthetic microbenchmarks execution time and network traffic, normalized to SMG. 103

Chapter 1

Introduction

1.1 Motivation

In nearly all compute domains, systems increasingly rely on parallelism and hardware specialization to exceed the limits of single core performance. GPUs, FPGAs, and other specialized devices are being incorporated into everything from mobile devices to supercomputers and data centers, offering new challenges and opportunities to both hardware and software designers. Arguably the most important challenge facing emerging architectures surrounds the efficient movement of data.

In many heterogeneous architectures, the accelerator is loosely-coupled with the host processor and used only to offload coarse-grain tasks with regular data access patterns and infrequent synchronization. A programmer must select a task for acceleration, port it to run on a single accelerator, and explicitly move any data that may be accessed to and from the accelerator's memory space before and after executing the task. This type of architecture avoids the need for costly coherence integration, and for dense regular applications, it can offer well-tuned execution, predictable performance, and efficient hardware utilization.

However, this simple approach lacks generality. Data must be explicitly copied between memory spaces, so workloads with irregular access patterns suffer from wasteful data movement. Synchronization requires conservatively flushing local memories, so applications with fine-grain tasks incur high communication costs. Accelerator memories are not coherent with the rest of the system, so workloads with unpredictable inter-task locality are unable to exploit reuse. This lack of flexibility is a significant and growing problem for two reasons. First, as programmers run out of dense coarse-grain tasks to accelerate, Amdahl's law has motivated a push to deliver efficiency gains for a much broader range of workloads. Applications with irregular sharing patterns, unpredictable control flow, and input-dependent locality (e.g. from the graph analytics, sparse matrix

algebra, or sparse machine learning domains) are now targets for acceleration. Irregularity and unpredictability can also lead to load imbalance, which motivates fine-grain synchronization for load balancing mechanisms (and further unpredictability). Second, growth in compute throughput is outpacing growth in memory throughput, creating a memory bottleneck that makes efficient on-chip data movement even more important. Explicit communication through main memory is highly inefficient and suboptimal, especially when that communication is wasteful due to workload irregularity.

A tightly-coupled accelerator architecture with a unified coherent address space can ease these burdens, enabling greatly improved programmability and communication efficiency for emerging irregular workloads. With a unified coherent memory space, programmers do not need to know in advance where each task will be computed or which data a task will access. Tasks can be dynamically scheduled, load balancing can be performed on-the-fly, data can be communicated only when needed, and devices can exploit implicit reuse in local memory. This explains why many heterogeneous architectures are moving towards tighter host and accelerator integration [30, 31, 86, 108]. At the same time, specialized storage technologies such as scratchpad memory, hardware queues, NVRAM, and high bandwidth 3D stacked DRAM are being incorporated with specialized compute technology in heterogeneous systems to improve performance. Similarly, enabling coherence in these specialized structures is important to offering portability, usability, and flexibility for future workloads. We envision a future where coherent heterogeneous systems can be rapidly prototyped and customized by building upon a simple coherence framework that can flexibly integrate multiple specialized devices with specialized storage across one or more systems-on-chip (SoCs), illustrated in Figure 1.1.

However, there are many challenges to implementing coherence efficiently in any heterogeneous system. In particular, a system designer must optimize two choices in tandem: 1) which coherence and consistency strategy should be used at each individual device or data storage element in the heterogeneous system, and 2) how to efficiently interface the diverse coherence strategies of these heterogeneous devices.

The first challenge, finding the ideal coherence and consistency strategy for a device in a heterogeneous system, will depend heavily on the memory demands of that device. For example, the memory demands of a GPU core differ significantly from those of a CPU core, motivating a dif-

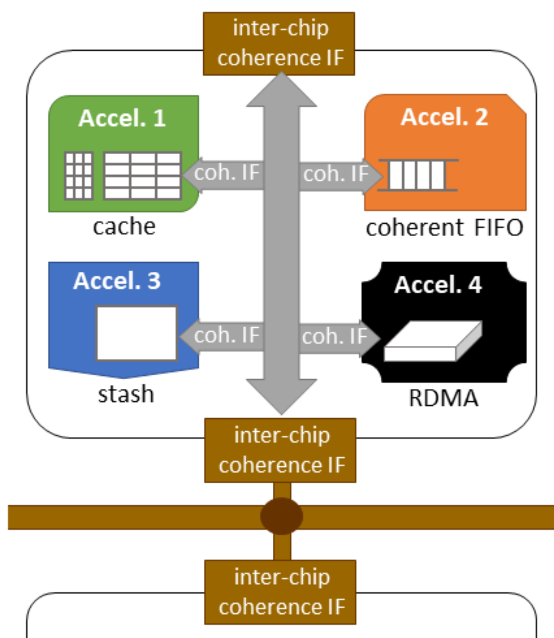


Figure 1.1: Coherence interface in a future heterogeneous system.

ferent coherence solution. CPU applications tend to prefer low latency memory accesses, and as a result they use hardware cache coherence protocols like MESI. Although complex, such protocols are effective at exploiting reuse and reducing latency for many CPU applications. However, unlike CPU applications, traditional streaming GPU applications are throughput-oriented. Thus, the complexity and overheads of MESI are a poor fit for these applications [103]. Instead, today's GPU caches use a software-based coherence protocol that is designed to be simple and lightweight. We refer to this strategy as GPU coherence, and it works by self-invalidating cached data and flushing (writing through) dirty data at synchronization points. This solution avoids the overheads and complexity of MESI-based protocols, and it offers good performance when synchronization is infrequent. However, as GPUs are used to accelerate a broader range of parallel applications that require more frequent synchronization, these invalidation and flush actions can degrade performance significantly [32, 38, 53, 61, 91].

This performance degradation has motivated two significant developments in GPU memory consistency models to allow efficient GPU coherence in the presence of frequent synchronization: scoped synchronization [55] and relaxed atomic semantics [52]. Both of these developments can greatly reduce the overheads of synchronization in GPU caches, but they both also add significant

complexity to the commonly used data race-free (DRF) memory consistency model. A DRF memory model offers sequentially consistent (SC) semantics to any program that properly synchronizes between communicating threads. Scoped synchronization additionally requires the programmer to be aware of the relative locations of communicating threads, which may be difficult or impossible depending on the program. Relaxed atomics remove the guarantee of sequential consistency, opening the door to behavior that is difficult to formally specify. The challenges of atomic relaxation have been thoroughly studied for CPUs and are no less relevant in the realm of GPU computing; even when used sparingly by expert programmers, their poorly defined semantics make it difficult to prove that code using relaxed atomics is correct.

The second challenge, how to best interface specialized devices in a coherent heterogeneous system, is difficult because of the widely varying memory demands and coherence strategies used by these devices. Strategies can differ in how they invalidate stale data (writer-invalidate vs. self-invalidate), in how they propagate dirty data (ownership vs. write-through caches), and in the granularity of state and communication (word vs. line). A heterogeneous coherence interface may need to support multiple strategies because no single strategy is able to support the memory demands of the devices being integrated (e.g. CPU and GPU), or simply because the accelerator IP being integrated comes from different vendors that use different protocols. Even within a single device, being able to dynamically optimize the coherence strategy to best fit the executing workload could improve efficiency in future systems. Whether the goal is functionality, composability, or performance, protocol flexibility is important in any heterogeneous coherence interface.

Although there have been many efforts in both academia and industry to efficiently integrate heterogeneous coherence strategies, proposed architectures tend to extend a fixed-granularity MESI-based last level cache (LLC), often using a hierarchical cache structure. Further, most proposed innovations rely on assumptions about sharing patterns in heterogeneous applications. In particular, they assume that sharing is either predominantly hierarchical (i.e., threads can be divided into groups that mostly communicate with each other), at a coarse granularity, or rare and exhibiting minimal locality. As heterogeneous coherent memory becomes more common and as specialized hardware is used to accelerate a more diverse range of applications, these approaches are unable to generalize to a broader range of sharing patterns. A more flexible solution is needed that is able to adapt to efficiently support devices with diverse synchronization frequency, spatial and temporal

locality, and throughput and latency requirements.

1.2 Contributions

Motivated by the lack of simple and flexible solutions to the above challenges, the innovations summarized below represent a principled framework for implementing efficient heterogeneous on-chip coherence based on simple consistency models that can flexibly and composably integrate the expected diversity of accelerators in future heterogeneous systems. Chapters 2-4 focus on optimizing the coherence and consistency strategy for an individual device, dealing specifically with the unique challenges of conventional caches in high-throughput devices such as GPUs. Chapters 5-6 describe a flexible coherence interface that can be used to integrate diverse coherence protocols on a single chip. In addition, Section 8.2 discusses how the innovations described in this thesis could be applied to benefit a broader range of system architectures and enable future optimizations.

All of the following work has been a collaborative effort. Matthew Sinclair is the lead author on the work described in Section 3.2 and Chapter 4. I am the lead author of the work described in Section 3.3 and Chapters 2, 5, and 6.

1.2.1 GPU Stall Inspector: Performance Characterization in High-Throughput Accelerators

As systems increasingly rely on highly parallel accelerators such as GPUs for performance gains, understanding latency bottlenecks in these systems becomes very important to hardware and software development. The first contribution, GPU Stall Inspector (GSI), addresses this need by identifying and classifying stall causes in highly parallel GPU workloads. Unlike prior GPU performance analysis tools, GSI focuses on characterizing the sources of memory stalls. This is of particular importance to designers of heterogeneous systems in which the memory system is the limiting factor for many emerging workloads. With GSI, a system designer can gain insight into the performance bottlenecks of emerging heterogeneous applications and coherence strategies, identifying detailed causes of delays due to synchronization, memory system latency limitations, and memory system throughput limitations. This work originally appeared in ISPASS 2016 [12]. The insights gained from GSI played an important role in developing the following research.

1.2.2 Efficient GPU Coherence without Scopes

While scoped synchronization has emerged as a popular mechanism for efficient heterogeneous coherence, particularly for GPUs, it complicates the DRF memory model and places an undesirable burden on the programmer while also limiting efficient communication patterns.

Chapter 3 describes two techniques to enable efficient coherence for high-throughput devices without complicating the memory model with scoped synchronization. The first technique proposes and evaluates the use of the DeNovo coherence protocol [42], originally proposed for multicore CPUs, to significantly improve data reuse for high-throughput accelerators such as GPUs. The DeNovo protocol obtains ownership for written data rather than writing it through at synchronization points. Since owned data does not need to be invalidated at synchronization points, locality can be exploited in this data even while preserving a relatively simple (scope-free) DRF memory model. This work originally appeared in MICRO 2015 [100] and received an honorable mention in IEEE Micro Top Picks 2016.

The next technique to improve heterogeneous coherence also aims to achieve the benefits of scoped synchronization without adding complexity to the DRF memory model or requiring the GPU cache to support ownership for data writes. Heterogeneous Lazy Release Consistency (hLRC) combines ideas from the DeNovo protocol, remote scope promotion [91], and lazy release consistency for distributed CPUs [60] to automatically detect when a thread is synchronizing locally, and to incur targeted cache invalidation and flush actions only when synchronization may be remote. hLRC represents another way to offer simple and efficient coherence in high-throughput accelerators such as GPUs. While DeNovo exploits locality in stores and atomic accesses to reduce the cost of synchronization in self-invalidating caches, hLRC avoids flush and invalidate actions entirely by exploiting synchronization locality. This work originally appeared in MICRO 2016 [10].

1.2.3 DRF-Relaxed: Atomic Relaxation with SC-Centric Semantics

Relaxed atomics are another popular way to reduce the high costs of synchronization in high-throughput accelerators, even though existing memory models do not provide clear semantics of program behavior when they are used. This work describes the DRF-Relaxed (or DRFrlx) memory model, which reconciles this problem by classifying common use cases of relaxed atomics and

defining the conditions for SC-centric execution when using each relaxation type [101]. In this way, DRFrlx enables the benefits of relaxed atomic performance for high-throughput accelerators while preserving SC-centric semantics. This work originally appeared in ISCA 2017 [101].

1.2.4 Spandex: Efficiently Integrating Heterogeneous Coherence Strategies

With Spandex we address the challenge of integrating the diverse coherence strategies that exist in heterogeneous systems. Spandex identifies key orthogonal dimensions in accelerator coherence design, defines a flexible request interface that can flexibly express these dimensions, and presents an integration protocol that efficiently and composably supports these requests. Rather than extending a coherence protocol designed for CPU memory demands, Spandex builds upon the hybrid hardware-software DeNovo protocol, which has been shown to offer scalability and efficiency for both CPU and GPU systems. The end result is an interface that is able to adaptively integrate whichever coherence strategies may be appropriate for future accelerators, while avoiding the complexity and scalability limitations of conventional MESI-based integration solutions. This work originally appeared in ISCA 2018 [11].

1.2.5 Dynamic Coherence Specialization

Flexible coherence interfaces like Spandex create new opportunities for improving coherence efficiency based on the dynamic demands of an executing workload. This section describes memory system optimizations that take advantage of locality, dataflow, and granularity hints that may be available from specialized programming languages, compilation tools, or hardware predictors to improve performance without complicating the DRF memory model. In implementing these extensions we build upon the flexibility and simplicity of the Spandex interface, as well as the insights gained from the analysis and optimization of heterogeneous coherence for emerging applications.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 describes how GPU Stall Inspector characterizes memory delay for high-throughput accelerators and demonstrates how it can be used to evaluate and motivate architectural changes in heterogeneous memory systems. Chapter 3 presents DeNovo

for GPUs and hLRC, two strategies for simple and efficient coherence in GPUs without the need for scoped synchronization, and compares them against traditional scope-based solutions. Chapter 4 formalizes the DRFrlx memory model and demonstrates how it enables the benefits of relaxed atomics without adding significant complexity. Chapter 5 defines the Spandex coherence interface and evaluates it against a more conventional hierarchical MESI-based protocol. Chapter 6 describes and evaluates optimizations to the Spandex interface that exploit hints about workload memory demands to dynamically optimize memory system performance. Chapter 7 highlights related work, and Chapter 8 summarizes the contributions of this work and discusses potential future research directions.

Chapter 2

GPU Stall Inspector: Performance Characterization in High-Throughput Accelerators

Although a common shared memory space offers significant advantages to emerging specialized architectures, the most efficient way to implement coherence in tightly coupled heterogeneous systems is still an open research question. Detailed performance characterization tools are therefore needed to help understand the differences between new heterogeneous coherence strategies, and to guide the development of heterogeneous coherence for future systems and workloads.

Performance characterization in high throughput accelerators such as GPUs is particularly difficult due to the high degree of parallelism. GPU cores can execute hundreds of threads concurrently, so attributing program delay to any single hardware or software element is difficult. While coarse-grained performance metrics such as total execution time may be sufficient for traditional GPU applications where latency attribution is fairly uniform across threads, more detailed information is needed to identify memory performance bottlenecks in less regular workloads. There have been previous efforts to provide more detailed performance profiling metrics [17, 15, 90, 72]; however, this prior work primarily targets discrete (loosely coupled) GPUs and focuses on the causes of delay within the GPU core. [1]

To help understand the detailed sources of memory delay in a tightly coupled CPU-GPU system, we introduce GPU Stall Inspector, or GSI. GSI does more than classify stalls based on the high-level cause of the pipeline delay available at the issue stage. It further subclassifies memory stalls based on the root cause of the memory delay. GSI is able to accomplish this because it is built upon an integrated CPU-GPU simulator based on Multifacet GEMS [82] and GPGPU-Sim. GEMS offers a detailed memory system model (Ruby) which is the source of this additional information.

[1] A more detailed description of related work is provided in Section 7.1.

2.1 Stall Classification in GSI

Understanding causes of GPU latency is complicated by the massive parallelism present in GPUs. A GPU consists of multiple cores, each of which can simultaneously execute hundreds of threads that process a highly parallel kernel program. Threads in a GPU are grouped into warps, which proceed together in lockstep through the pipeline. Warps are grouped into thread blocks, and warps in the same thread block are guaranteed to be executed on the same core. In each cycle, a GPU may issue one instruction per issue slot from its warps. We define a stall cycle as any cycle in which no warp instructions are issued by a GPU core. Section 2.1.1 describes the various stall types that may prevent a warp instruction from issuing.

In each stall cycle there may be multiple different warp instructions that are stalled for multiple different reasons. In Section 2.1.2 we describe how a single stall type is chosen from among these stall causes and attributed to each stall cycle. Finally, since we are most interested in memory stalls, in Sections 2.1.3 and 2.1.4 we subclassify memory stalls further based on their underlying causes.

2.1.1 Classifying Causes of Instruction Stalls

When an instruction can be issued in a cycle, the cycle is classified as **no stall**.

An **idle stall** occurs when there are no active warps available to issue instructions.

A **control stall** occurs when the instruction supplied by the instruction buffer is not the next instruction to be executed in a warp. If control stalls dominate, there is significant divergence in the kernel code.

A **synchronization stall** occurs when a warp is blocked due to a pending synchronization operation (acquire, release, or thread barrier). GPUs use acquire and release operations to preserve memory consistency and thread barriers to synchronize threads in a thread block.

A **memory data stall** occurs when an instruction cannot issue because it is dependent on the result of a pending load. Section 2.1.3 provides more details on the different subcategories of memory data stalls.

A **memory structural stall** occurs when a memory instruction is unable to issue to the load/store unit because it is full. There are multiple sources of memory structural stalls, which we

describe in more detail in Section 2.1.4.

A **compute data stall** occurs when an instruction cannot issue because it is dependent on the result of a pending compute (non-memory) instruction.

A **compute structural stall** occurs when a compute instruction cannot issue because the appropriate compute unit is occupied.

2.1.2 Attributing Stalls to Cycles

GSI stall cycle attribution proceeds in two stages. First, a single stall type is assigned to each warp instruction considered in the issue stage. When classifying the stall cause assigned to an individual warp instruction, priority is given to the stall cause that is most strongly preventing execution. Intuitively, the strength of a stall cause is linked to the likelihood that the instruction will remain stalled in the next cycle. This is not always straightforward to define, and should be tailored to the profiling goals. We specify the prioritization used in Algorithm 1.

After each instruction considered in the issue stage has been classified, the issue cycle itself is classified based on the stall causes of the individual warp instructions. The prioritization we use to classify cycle stall causes is defined in Algorithm 2. Generally, the cycle stall classification is set to match the weakest stall cause found. Intuitively, this is the stall cause of the instruction that was closest to issuing. Since this must be the strongest stall cause for that instruction, removing that stall cause will likely improve performance. However, removing the stall cause may not cause a proportional reduction in the stall count for the removed stall type. A single stalled instruction can be blamed for many stall cycles even if all other instructions are unable to issue for unrelated reasons. Thus, removing the original stall cause and allowing this instruction to issue could simply cause other stall types to dominate.

The "weak" cycle stall classification priority described in Algorithm 2 is not an exact inversion of the "strong" instruction stall classification priority. Memory stalls and synchronization stalls are prioritized over compute stalls in both classification algorithms because we are interested in analyzing the effects of changes to the memory system.

Algorithm 1 Instruction Stall Classification

```
if No active warps to consider then
    classify idle stall
else if The next instruction to issue is unavailable then
    classify control stall
else if Warp is blocked for a synchronization then
    classify synchronization stall
else if Instruction has a data hazard on a pending load then
    classify memory data stall
else if Instruction has a structural hazard on load/store unit then
    classify memory structural stall
else if Instruction has a data hazard on a pending compute operation then
    classify compute data stall
else if Instruction has a structural hazard on a compute unit then
    classify compute structural stall
else if Instruction is able to issue then
    classify no stall
end if
```

2.1.3 Sub-Classifying Memory Data Stalls

Memory data stalls are subclassified based on where the dependency load was serviced. The sub-categories are: **L1 cache**, **L1 coalescing**, **L2 cache**, **remote L1 cache**, and **main memory**.

L1 cache stalls mean that instructions have been stalled because they are dependent on loads that are satisfied locally. This can happen if data is used immediately after it is loaded or if there is a delay in the load/store unit. L1 coalescing stalls are due to requests that have missed in the L1 cache but were satisfied by a response for another request to the same line. L2 cache stalls are due to dependencies for requests that are satisfied in the L2 and may mean that the L1 cache is not being efficiently utilized or that the data access pattern does not allow for reuse at L1. Remote L1 cache stalls are caused by dependencies for data requests that are satisfied at a remote L1 core. These are only possible in protocols like DeNovo [110] that enable ownership in L1 caches. Main memory stalls indicate dependencies on accesses to main memory and can occur when the data set is too large to fit in the L2 cache or the L2 is not being utilized efficiently.

2.1.4 Sub-Classifying Memory Structural Stalls

Memory structural stalls occur when a ready memory instruction is blocked from issuing to the load/store unit. Possible causes depend on the memory system being studied, but they are often

Algorithm 2 Issue Cycle Stall Classification

```
if At least one instruction was able to issue then
    classify no stall
else if At least one memory structural stall was found then
    classify memory structural stall
else if At least one memory data stall was found then
    classify memory data stall
else if At least one synchronization stall was found then
    classify synchronization stall
else if At least one compute structural stall was found then
    classify compute structural stall
else if At least one compute data stall was found then
    classify compute data stall
else if At least one control stall was found then
    classify control stall
else if At least one idle stall was found then
    classify idle stall
end if
```

due to multiple pending memory accesses. For the memory system configurations considered in this work, memory structural stalls can be caused by a miss status handling register (MSHR) running out of entries (**full MSHR**), a **full store buffer**, a **bank conflict**, a **pending release**, or an active direct memory access (DMA) operation that is blocking subsequent requests (**pending DMA**).

Memory structural stalls due to a full MSHR may indicate there is bursty load miss traffic or the MSHR is too small. Similarly, stalls due to a full store buffer may mean there is bursty store miss traffic or the store buffer is too small. Bank conflict stalls can occur if data accesses are not evenly strided across cache or local memory banks. In the system studied, release synchronization operations block stores from issuing until all prior stores are flushed, causing pending release stalls. Similarly, a memory instruction will be blocked if it is trying to access the result of a bulk direct memory access operation (DMA) before the operation is complete, resulting in pending DMA stalls.

2.2 Case Study: Stash vs. Scratchpad+DMA

To demonstrate the value of GSI we use it to evaluate two potential improvements to the scratchpad memory structure used in modern GPUs. We compare the performance of these innovations against a baseline scratchpad implementation and analyze the stall breakdowns of these two innovations

for a single application. Using this information we are able to determine the precise causes of performance degradation and motivate changes to the hardware configuration.

2.2.1 Background

Scratchpad memory is a directly addressed memory space that is not kept coherent and is private to a thread block. The scratchpad can be used to cheaply satisfy data accesses that would otherwise need to use the cache. Because all threads in a thread block share the space, scratchpad memory is often used for efficient intra-thread block communication as well.

Despite these benefits, scratchpads suffer from multiple inefficiencies. Typical scratchpad use involves loading data into the scratchpad at the start of a kernel, performing computations on the data in the scratchpad, and then copying the scratchpad data to global memory at the end of the kernel. In this scenario, copying data between the global memory and the scratchpad pollutes the cache and registers and increases instruction count and energy. In addition, the lack of coherence requires that any data that could potentially be accessed by other cores must be conservatively transferred to and from the scratchpad. Thus, if a memory region is only accessed sparsely and unpredictably, the entire region must be copied to and from the scratchpad memory, incurring waste. Because of these inefficiencies, only kernels that access a memory region multiple times in a predictable fashion tend to benefit from using scratchpads.

D2MA [57] addresses the inefficiencies of scratchpads by offloading the process of loading data into the scratchpad to a separate DMA. The DMA engine explicitly transfers data into the scratchpad in bulk without polluting the L1 cache or registers. At the start of a kernel a mapping from scratchpad to global data is defined. Scratchpad memory accesses to the mapped region are then blocked (on the first use) until the entire DMA transfer is complete. We approximate D2MA by adding a DMA engine to the baseline scratchpad memory. This implementation, which we refer to as scratchpad+DMA, differs from D2MA in that the DMA engine can transfer data to and from scratchpad (D2MA only loads data into the scratchpad) and a load to an incomplete DMA mapping blocks progress at a core granularity rather than a warp granularity.

Stash is a hybrid memory structure that addresses the inefficiencies of scratchpad accesses and cache accesses by making scratchpads a part of the coherent global address space [63]. The stash stores a mapping from local to global data (and the reverse) in a stash map structure. When any

address in the stash mapping is first accessed, the stash map is used to generate a global request to the memory system. When the data returns, it bypasses the cache and is directly loaded into the stash. Subsequent accesses to the mapped address will always hit locally and return without translation. The stash map also enables a GPU core to respond to global requests for data that is dirty in the stash. These coherence mechanisms allow stash data to be loaded on-demand and to be lazily written back, which is not possible with D2MA. This avoids wasteful copies and wasteful instructions involved in managing a scratchpad.

All three configurations use DeNovo coherence [42] for GPU and CPU caches and for the stash (described in detail in Section 3.2). Both scratchpad+DMA and stash improve upon scratchpad memory because the data transfers to and from scratchpad/stash memory bypass the cache. The stash also implicitly transfers data between the local and global address spaces. The main difference between stash and scratchpad+DMA is that stash data is kept coherent with the shared global memory. This means that stash accesses can generate global load requests on-demand and that dirty data can be lazily written back because it is globally visible. In contrast, scratchpad with DMA must conservatively transfer all data into the scratchpad at the start of a kernel, and conservatively write back any potentially modified data at the end of the kernel.

The DMA data movement pattern can be helpful or harmful. Bulk reads and writes happen in parallel and can be a very effective preloading technique for avoiding the memory latency of successive on-demand load misses. However, scratchpad accesses to a pending DMA allocation must stall until the entire DMA transfer completes. This latency combined with the congestion caused by bursty DMA traffic may negatively affect performance.

The tradeoffs of stash memory and scratchpad+DMA result in differences in the memory system that can reduce some sources of performance degradation but increase others. We use GSI to better understand how these differences affect GPU performance, focusing on one microbenchmark: Implicit.

Implicit is a synthetic microbenchmark used by Komuravelli, et al. to evaluate the benefits of stash [63]. In the Implicit microbenchmark, an array of data is allocated and mapped to scratchpad/stash memory. Each thread block is assigned a chunk of the array. Each thread reads an element in the chunk, performs a computation on the element, and writes the result back to the same location in memory.

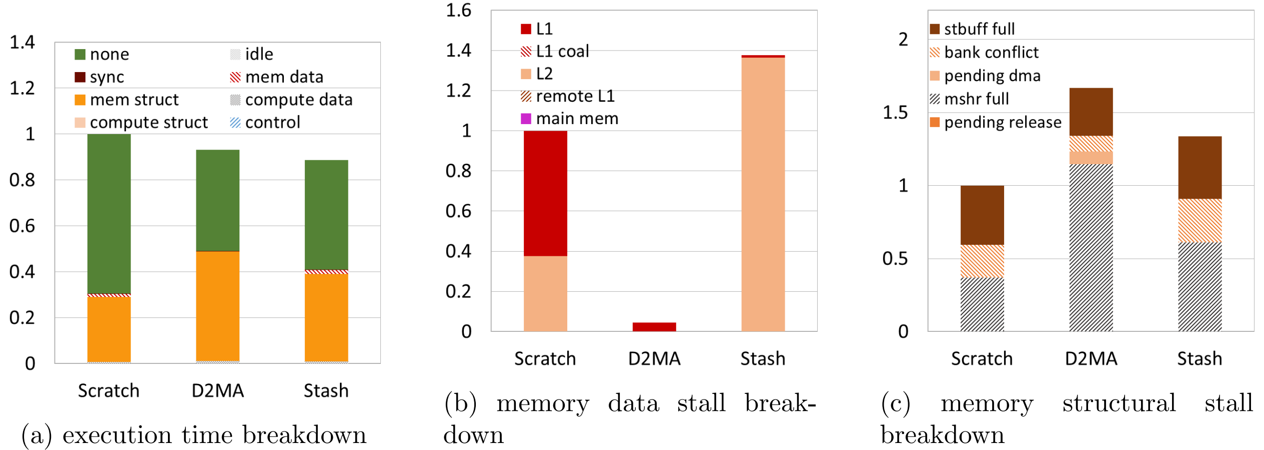
Table 2.1: Parameters of the simulated heterogeneous system.

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
GPU cores used	1
Scratchpad/Stash Size	16 KB
Number of Banks in Stash/Scratchpad	32
Memory Hierarchy Parameters	
L1 and Stash/Scratchpad hit latency	1 cycle
Remote L1 and Stash hit latency	35−83 cycles
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

This microbenchmark highlights the advantage of implicitly loading data into scratchpad/stash memory and is representative of applications that access data in a regular streaming manner. Stash implicitly moves data from global to scratchpad/stash memory while scratchpad+DMA uses a DMA engine to reduce the overheads of explicit data movement, so both configurations should improve performance over the baseline scratchpad configuration. By applying GSI to this application and analyzing the stall breakdown, we can understand the impact of each of the above effects on overall performance, and we use this information to motivate a change in the hardware system used.

2.2.2 Methodology

We use GSI to compare D2MA and stash storage strategies for the Implicit microbenchmark. GSI is implemented on top of an integrated CPU-GPU architectural simulator. The Simics [78] full system simulator models the CPU cores, GPGPU-Sim [17] models the GPU cores (the modeled GPU is similar to an NVIDIA GTX 480), the Wisconsin GEMS memory timing simulator [82] models the memory system, and Garnet [7] models the interconnect. The system contains 1 CPU core and 15 GPU cores distributed across a 4x4 mesh network. Each core has a private L1 and all cores share a banked last level L2 cache. The simulated stash memory is identical to that used in the original stash publication [63]. For simplicity, only one GPU core is utilized in the evaluation of the Implicit microbenchmark. Table 2.1 summarizes the key architectural parameters of our simulated



(a) execution time breakdown

(b) memory data stall breakdown

(c) memory structural stall breakdown

Figure 2.1: Stall cycle breakdowns for implicit microbenchmark (normalized to baseline scratchpad).

system. A similar simulation setup is also used for some subsequent work in this thesis. While past studies have validated individual components of this simulator, the full integrated simulator differs from existing hardware in its flat integrated cache structure, and as a result no full system validation of this integrated simulator has been attempted. The changes required to implement GPU stall profiling are minimal. GSI increases simulation time by on average 5% for a set of representative workloads, with this proportion decreasing as input sizes increase. The introduction of GSI does not affect simulation results, and all results are deterministic.

2.2.3 Evaluation

Figure 2.1a shows the GPU execution time breakdown of the implicit microbenchmark. Figures 2.1b and 2.1c show the memory data stall and memory structural stall subclassifications, respectively. We use these results to identify causes of performance disparity between stash and scratchpad+DMA memory configurations.

Figure 2.1a shows that the number of "no stall" cycles is reduced by 36% and 31% for scratchpad+DMA and stash, respectively. This occurs because both the scratchpad+DMA and stash configurations reduce instruction count. The scratchpad+DMA configuration does this by offloading the loads to its DMA engine, and the stash does this by loading mapped data on-demand from global memory into the stash. Thus, neither configuration needs initial scratchpad load and store instructions that pollute the registers or L1 cache. However, the reduction in "no stall" cycles is not uniform, and is offset by increases in other stall types. This kind of benefit offset is not uncommon

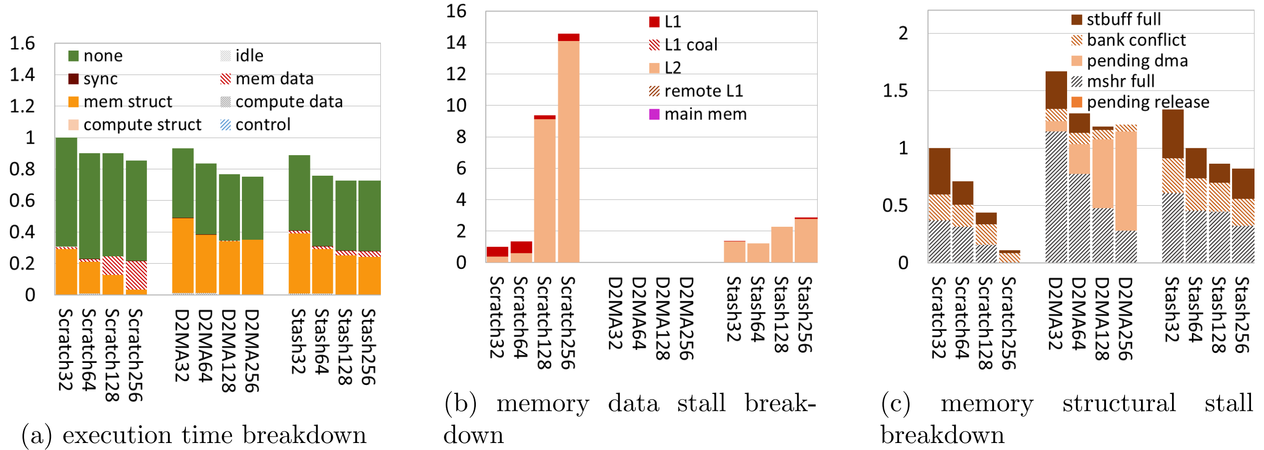
when profiling stall causes.

In the case of the Implicit microbenchmark, the “no stall” cycle reductions in scratchpad+DMA and stash are offset primarily by 67% and 34% increases in memory structural stalls, respectively. Scratchpad has fewer memory structural stalls because the explicit transfer of scratchpad data serves to restrict the rate of requests to the memory system. At the start and end of the kernel, the baseline scratchpad implementation issues many concurrent loads and stores (respectively) to the memory system, incurring memory structural stalls due to bank conflicts, a full store buffer and a full MSHR. However, these memory operations are interleaved with instructions that perform address calculations and scratchpad store instructions, so the rate at which global memory loads are issued to the memory system is limited. As a result, the number of memory structural stalls is also limited.

In comparison, scratchpad+DMA does not need to process any data transfer instructions and is able to issue load requests to the memory system as fast as the DMA engine allows (one per cycle). This increased request rate causes the MSHR to fill up faster. These generated requests bypass the pipeline and the cache, so bank conflicts are insignificant, but as soon as a normal memory access tries to issue to the global memory or scratchpad memory, it is blocked due to a full MSHR or a pending DMA (only for dependent instructions).

Stash generates a global memory access at the first load of each stash data word. Since the stash is directly addressed, fewer instructions are needed to compute stash addresses than are needed to compute the target address for requests to global memory in the baseline scratchpad configuration. This means fewer instructions are interleaved with the memory instructions, and memory instructions are issued to the memory system at a faster rate than for baseline scratchpad. This increased memory request rate increases the amount of memory structural stalls due to a full MSHR, a full store buffer, and bank conflicts.

Overall, the stall breakdowns tell us that scratchpad+DMA and stash are able to improve performance, but the amount of time saved is offset by additional stalls caused by increased memory request frequency. Furthermore, although there is a difference in data stalls, it is insignificant.



(a) execution time breakdown

(b) memory data stall breakdown

(c) memory structural stall breakdown

Figure 2.2: Stall cycle breakdowns for implicit microbenchmark with varying MSHR sizes (normalized to baseline scratchpad with 32-entry MSHR).

2.2.4 MSHR Sensitivity

Since MSHR stalls are significant for this microbenchmark, we investigate the effect of increasing MSHR size on performance. We run the Implicit microbenchmark on all three configurations while increasing MSHR size from 32 to 256. We also scale the store buffer size with the MSHR size to prevent store buffer stalls from becoming the new bottleneck. Figure 2.2a shows the results normalized to baseline scratchpad using an MSHR with 32 entries.

The decrease in full MSHR stalls shows that increasing MSHR size benefits all configurations. However, each configuration benefits from the increased MSHR size in different ways. A 256-entry MSHR completely eliminates full MSHR stalls for the baseline scratchpad configuration, but memory data stalls significantly increase (13X compared to its original value). Memory data stalls increase for the scratchpad configurations because the instruction following the explicit load (a store of the value to scratchpad memory) is dependent on the result of that load. Although many more explicit loads are able to issue with a larger MSHR, the next instruction will need to block until the load has completed.

As MSHR size increases, MSHR stalls decrease for the scratchpad+DMA configuration. However, memory structural stalls due to pending DMA requests significantly increase as MSHR size increases (8.9X more with a 256-entry MSHR). These memory structural stalls increase because the first scratchpad accesses occur early in the application, which causes threads to be blocked due to a pending DMA soon after the application begins.

Increasing MSHR size also reduces MSHR stalls for the stash configuration by allowing a greater number of threads to concurrently issue memory requests (implicitly in the case of the stash configuration). Although MSHR stalls decrease, memory data stalls increase (2.1X compared to the original value) because the issued requests have not completed before a dependent instruction needs to use the returned value. However, the increase is less significant for stash compared with scratchpad. This is because the scratchpad implementation must wait for all loads to the scratchpad to complete before proceeding to the compute phase. The stash configuration, on the other hand, will issue loads on-demand within the compute phase. Although many warps will still be blocked waiting for the result of a load to stash memory, there are more active threads during the compute phase that can utilize the issue slots with useful work. As a result, stash achieves higher core utilization than scratchpad and experiences a smaller increase in memory data stalls as MSHR size increases.

Overall, these results show that increasing MSHR size improves performance for all configurations. However, alleviating this bottleneck causes other stall sources to increase. For the scratchpad configuration, increasing the number of MSHRs does not help as much because of data dependencies. Increasing MSHR size is more beneficial for the scratchpad+DMA configuration, which sidesteps the data dependencies by prefetching data. However, because the scratchpad data is used early in the application, memory structural stalls due to pending DMA requests increase for the scratchpad+DMA configuration. The stash configuration also benefits from increased MSHR size but is better able to utilize the core by only blocking scratchpad loads at the granularity of warps (as discussed in Section 2.2.1). Thus, these results demonstrate the benefits of the stash's hybrid memory organization.

2.3 Summary

GSI delivers a detailed breakdown of every stall cycle in the execution of a GPU application. This information can help to identify performance bottlenecks in emerging heterogeneous applications, confirm and explain unexpected heterogeneous simulation results, help understand performance differences between heterogeneous coherence strategies, and motivate changes to hardware and software to accelerate high-throughput workloads. As new strategies are proposed to enable efficient

heterogeneous coherence for emerging workloads with irregular sharing and synchronization, these insights only grow in importance.

Overall, GSI has been a crucial tool in the development of subsequent work in this thesis. Although GSI is focused specifically on stall causes in GPU systems, the concepts of stall cause prioritization and memory stall sub-classification can be applied to other types of highly parallel specialized hardware as well.

Chapter 3

Efficient GPU Coherence Without Scopes

Since coherence overheads and scalability limitations make MESI coherence a poor fit for GPU workloads, modern GPUs use a software-based coherence strategy, which we refer to as GPU coherence. Instead of tracking owners and sharers, GPU coherence triggers bulk cache invalidation and flush actions at specified points to keep caches coherent.

GPU coherence is simple to implement, but it can be very inefficient if the bulk flushes and invalidations are frequent. There have been many efforts to limit the impact of these actions, but perhaps the most common are scoped synchronization and relaxed atomics, both of which add significant complexity to the memory model. In the following sections we discuss the limitations of scoped synchronization and related techniques, and we propose two changes to the GPU *coherence protocol* that enable similar efficiency without requiring complex changes to the consistency model: DeNovo for GPUs (Section 3.2) and heterogeneous lazy release consistency (hLRC, Section 3.3). Chapter 4 discusses the challenges of relaxed atomics and proposes an extension to the DRF *consistency model* that offer simple semantics for practical use cases of atomic relaxation.

3.1 Background

In this chapter we describe existing strategies for efficient GPU coherence and consistency and propose simpler and more flexible alternatives. Table 3.1 summarizes the tradeoffs of each GPU coherence and consistency strategy discussed. Each strategy is classified by whether it is simple and scalable, whether it enables dynamic sharing (defined in Section 3.1.3), and what memory access properties it relies on to avoid flush and invalidation overheads and achieve efficient coherence.

Table 3.1: Tradeoffs of each GPU coherence and consistency strategy.

Coherence+ Consistency	**Simple semantics**	**Scalable**	**Efficient dynamic sharing**	**Avoids synch. overheads for:**	
				High synch. locality	**Locality in owned data**
GPU coh.+DRF	✓	✓	✓	✗	✗
GPU coh.+HRF	✗	✓	✗	✓	✗
GPU coh.+HRF+RSP	✗	✗	✓	✓	✗
DeNovo+DRF	✓	✓	✓	✗	✓
DeNovo+HRF	✗	✓	✓	✓	✓
hLRC+DRF	✓	✓	✓	✓	✗

3.1.1 Data Race Free Memory Models (GPU coh.+DRF)

Our baseline GPU coherence implementation assumes a relatively simple memory consistency model that guarantees sequentially consistent behavior for data race-free (i.e., properly synchronized) programs, also called DRF. Under the DRF memory model, defined by Adve and Hill [4], data accesses are distinguished from synchronization (or atomic) accesses in software. A program is data race-free if, in every SC execution of the program, any two conflicting data accesses from different threads are ordered by a happens-before relation, which is essentially a chain of paired synchronization accesses (e.g., the release and acquire of a lock protecting a critical section). If no such chain exists between conflicting data accesses from different threads, this constitutes a *data race*. A system is DRF compliant if every execution of a DRF program is guaranteed to be sequentially consistent. As long as bulk cache flush and invalidate actions are performed at every synchronization access (or kernel boundary, or memory fence), GPU coherence is compliant with the DRF memory model and can guarantee SC behavior for any DRF program.

Using DRF with GPU coherence combines simple semantics with a scalable protocol. However, it requires flush and invalidates to be performed at every synchronization action, which can be very inefficient if synchronization is frequent.

3.1.2 Scoped Synchronization (GPU coh.+HRF)

One common strategy to reduce the cost of synchronization in GPU caches is to use scoped synchronization. Scoped synchronization allows the programmer to avoid cache flushes and invalidations when they are known to be unnecessary. This can significantly improve performance, but it also adds a great deal of complexity to the memory consistency model. Despite this complexity, the notion of scoped synchronization is ubiquitous in GPU memory models and is present in both

the HSA memory consistency model [51] as well as (less precisely) in the CUDA programming specification [88], two of the most widely used GPU memory models in industry.

The semantics of scoped synchronization are formalized in the heterogeneous race-free (HRF) memory model [55], which builds upon the data race-free (DRF) memory model. HRF adds the notion of scope to each synchronization access. If two threads are known to be in the same thread block (and thus share a local cache, according to the programming model of GPU languages), they may synchronize using local scope. Since coherence actions like cache invalidations and flushes are not necessary for communication between threads that share a local cache, an HRF system does not need to perform flushes or invalidates for locally scoped synchronization.

The use of the HRF consistency model can improve efficiency for applications with high synchronization locality. However, the introduction of scoped synchronization leads to two important drawbacks: complex consistency semantics, and inefficient dynamic sharing.

HRF complicates the consistency model by introducing a new illegal race type: the *heterogeneous race*. In the HRF model, even if there is a paired synchronization between two conflicting accesses in different threads, they may still form a *heterogeneous race* if the incorrect scope is used (e.g., local synchronization is used between threads in different thread blocks). This requires the software to have static knowledge of the relative locations of communicating threads whenever scopes are used. If there is any uncertainty about whether a thread is communicating locally or not, locally scoped synchronization may not be used for that access.

Using GPU coherence+HRF also limits synchronization flexibility by preventing a system from exploiting locality in the presence of dynamic sharing. A dynamic sharing pattern is any sharing pattern where the identity of communicating threads is not statically known. For example, an application may use local task queues and work stealing to implement load balancing among a collection of worker threads. In this scenario, every worker processes local tasks until its queue is empty, at which point it may steal from any other worker's queue. Even though communication is primarily local, global scope must be used for all queue accesses to ensure correctness in an HRF system.

3.1.3 Remote Scope Promotion (GPU coh.+HRF+RSP)

Remote scope promotion (RSP), proposed by Orr et al. [91], addresses the problem of dynamic sharing in the HRF consistency model by allowing a thread to promote the scope of an access on a remote thread. In the work stealing example, threads could use local scope when accessing their local queue, and RSP when stealing from a remote queue. Even when communicating threads do not share a local scope, RSP triggers coherence actions that enable a thread to safely synchronize with a locally scoped access in a remote core.

However, RSP is far from a perfect solution. While it does give programmers added flexibility in when they may use local scope, RSP arguably makes the memory model even more complex. Programmers now have a new type of synchronization access to reason about, and they must still be aware of the relative locations of communicating threads to determine when RSP is necessary. In addition, the hardware implementation of RSP can be inefficient. To dynamically promote the scope of any potentially synchronizing access, RSP needs to conservatively perform bulk coherence actions at all remote caches. The overheads of triggering a cache flush or invalidation at every cache can seriously degrade performance if RSP is frequent, and this expensive broadcast operation limits RSP's scalability to high core counts.

3.2 DeNovo Coherence for GPUs (DeNovo+DRF)

To improve GPU coherence without adding complexity to the consistency model, we first propose the use of the DeNovo protocol for coherence in GPU caches. DeNovo is a scalable protocol that has the advantage of being able to exploit locality in written data across synchronization points. It is therefore able to avoid the complexities of HRF, offering simple semantics and natural support for efficient dynamic sharing.

3.2.1 DeNovo Coherence for Multicore CPUs

DeNovo is a hybrid hardware-software coherence protocol initially designed for multicore CPU systems [42, 111, 109]. Past work has shown that DeNovo improves scalability and efficiency in multicore CPUs by reducing coherence overheads and complexity relative to MESI [42, 111, 109]. DeNovo assumes a DRF memory model and self-invalidates stale data at synchronization points.

By using self-invalidation, DeNovo avoids a significant source of coherence overhead and complexity in MESI-based systems: sharer tracking, writer-initiated invalidation, and the associated transient states. Like MESI, DeNovo obtains ownership for written data and atomic accesses. Without the need for sharer tracking and invalidation, DeNovo ownership is much simpler than MESI ownership. Ownership is obtained for writes and atomic accesses, and the last level cache (LLC) tracks the ID of the owning core the data field, avoiding the need for separate directory storage. Requests for owned data are forwarded directly to the owning core, requiring no transient blocking states at the LLC. As detailed in the following section, the qualities that allow DeNovo to scale well in multicore CPUs make it an attractive scope-free option for high throughput devices such as GPUs as well.

3.2.2 Qualitative Insights

Both hardware-based MESI and software-based GPU coherence face serious limitations when it comes to implementing coherence in high-throughput caches like those used by GPUs. MESI suffers from high coherence overheads that scale poorly to high core counts and high throughput. GPU coherence incurs expensive coherence actions at synchronization, and the most common ways to reduce these costs (scoped synchronization, RSP) require adding significant complexity to the consistency model. To avoid these limitations, this work proposes and evaluates the use of the hybrid hardware-software DeNovo coherence protocol for GPU caches.

Compared with GPU coherence, DeNovo exhibits comparable simplicity while offering improved cache reuse. Both DeNovo and GPU coherence flush dirty data at release synchronization points or when the store buffer is full, but GPU coherence writes dirty data through to the LLC while DeNovo obtains ownership for any non-owned dirty data. DeNovo also obtains ownership for atomic accesses. Both protocols self-invalidate their caches at acquire synchronization points to avoid the complexity of writer-initiated invalidation, but DeNovo only needs to invalidate non-owned data.

Obtaining ownership involves both costs and benefits. Owned data is not invalidated or flushed at synchronization points, and all subsequent accesses from the same core will hit on that data until it is evicted or ownership is revoked by a remote requestor. This enables DeNovo to exploit much more cache reuse in written data and atomic accesses than GPU coherence. Even when locality in owned data is low, an owner device can respond directly to a remote request for data, avoiding a wasteful data transfer to and from the LLC. Thus, DeNovo avoids much of the overhead incurred

by flush and invalidate actions, which is particularly important for workloads that exhibit frequent synchronization. Relative to GPU coherence, DeNovo is able to exploit more locality in written data and atomics, reduce wasteful communication, and improve cache performance.

The costs of DeNovo ownership are relatively minimal. Because it tracks owned state at word granularity, DeNovo requires an extra state bit per word in the L1 cache and the LLC (3% overhead for a 32-bit word). No directory storage is needed since the owner ID for any owned word is stored in the data field of that word (this does, however, require that the LLC is inclusive for owned data). Obtaining ownership can also add indirection latency for a future remote request, which must look up the owner at the LLC. Overall, DeNovo ownership does incur some overhead relative to GPU coherence, but these costs are outweighed by its benefits: more cache reuse and less wasteful data transfer.

Importantly, DeNovo obtains these benefits without requiring changes to the simple DRF memory model. Compared with GPU coherence+HRF, DeNovo+DRF does not entirely avoid flush and invalidation actions when synchronization is known to be local, but it is able to improve cache efficiency when synchronization is unknown or global, offering improved simplicity, flexibility, and efficiency for emerging applications. The efficiency of a lock access in DeNovo+DRF, for example, is not dependent on statically available knowledge about which thread may next access the lock. Furthermore, if synchronization locality is known and an HRF memory model is assumed, DeNovo+HRF can be expected to outperform GPU coherence+HRF because it achieves the benefits of scoped synchronization while also offering efficient reuse in owned data when synchronization is not local.

3.2.3 Evaluation

We compare the performance of GPU coherence and DeNovo using DRF and HRF memory models for both coherence strategies. Five configurations are evaluated, listed below:

- **GD**: GPU coherence + DRF consistency.
- **GH**: GPU coherence + HRF consistency.
- **DD**: DeNovo coherence + DRF consistency (DeNovo+HRF in Table 3.1).

- **DD+RO**: DeNovo coherence + DRF consistency with an optimization that uses DeNovo regions (originally described by Choi et al. [42]) to avoid self-invalidating data marked as "read only" at synchronization points.

- **DH**: DeNovo coherence + HRF consistency (DeNovo+HRF in Table 3.1).

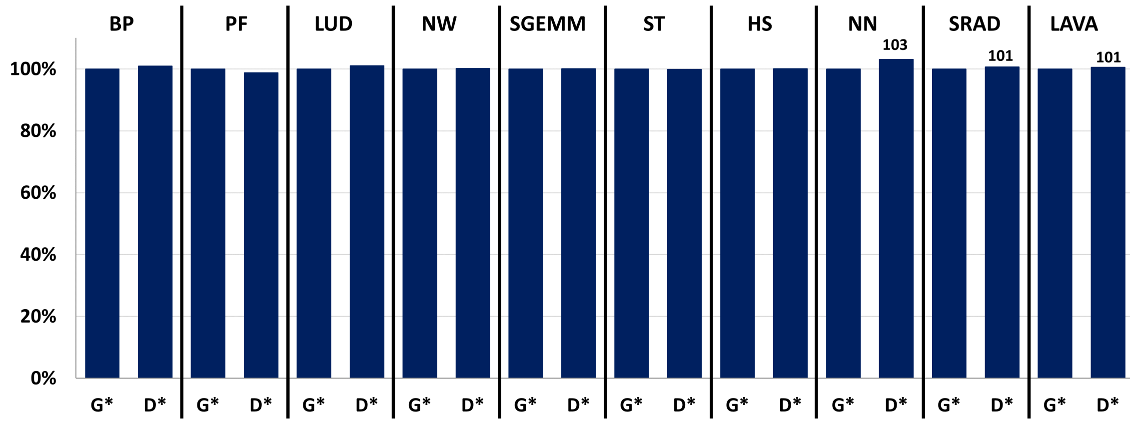
The workloads studied include real applications and microbenchmarks, summarized in table 3.2. The real applications are a set of conventional GPU benchmarks with no intra-kernel synchronization; the only synchronization is at the kernel boundary. They are used to verify that DeNovo does not harm performance for traditional applications where synchronization is infrequent. Since existing GPU applications rarely use intra-kernel synchronization, we adapt a set of simple microbenchmarks containing mutex locks, semaphores, and barriers from the SyncPrims benchmark suite developed by Stuart and Owens [107], [1] as well as an unbalanced tree search application used in the HRF work [55] to test synchronization efficiency. These microbenchmarks are divided into workloads that primarily use local synchronization and workloads that only use global synchronization. For all microbenchmarks, synchronization constructs are used to guard reads and writes to shared data. When synchronization is local, the protected data is partitioned and only accessed by threads in the same core, and when synchronization is global it is accessed by multiple cores.

These workloads and configurations uses the same integrated CPU-GPU architectural simulator as that used in Section 2.2 with a 4x4 mesh containing 1 CPU node and 15 GPU nodes. Dynamic energy consumption is measured using GPUWattch [73]. Detailed simulation parameters are summarized in table 3.3.

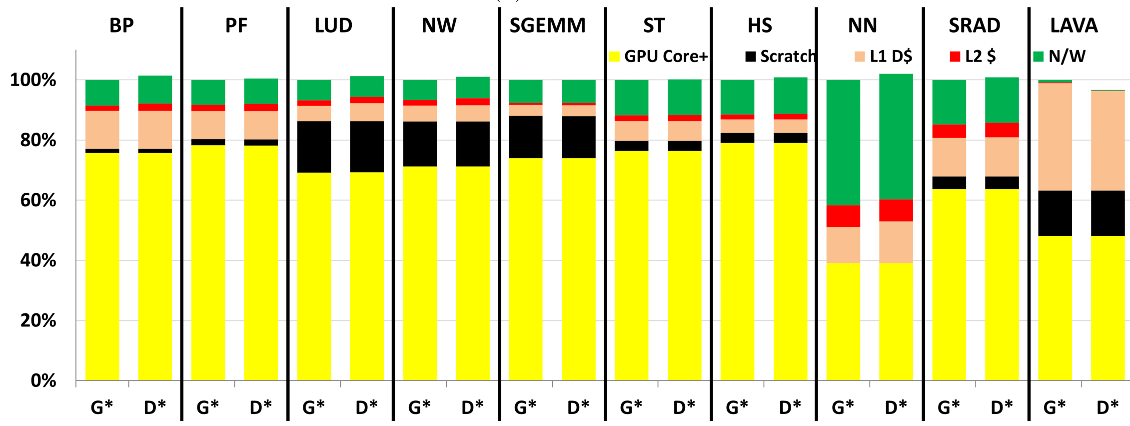
Figure 3.1 shows the execution time, dynamic energy, and network traffic of GPU coherence (G*) and DeNovo (D*) for the real applications, normalized to GPU coherence. Dynamic energy is divided into multiple components based on the source of energy (GPU core+,[2] scratchpad, L1, L2, and network), and network traffic is divided by message type. Figure 3.2 shows these results for microbenchmarks with only global synchronization. For both of these sets of benchmarks, the consistency model used is irrelevant since there are no opportunities for local synchronization.

[1]These adapted applications make up part of the Heterosync benchmark suite [102]

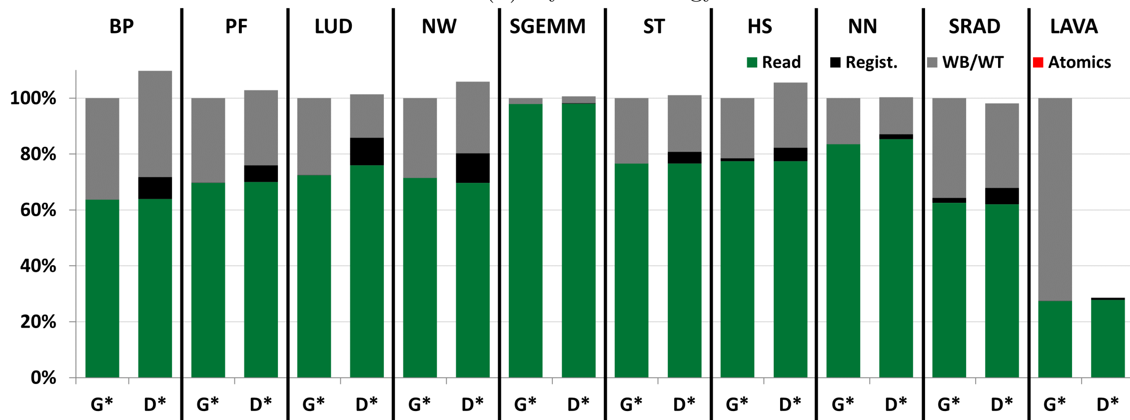
[2]GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, scheduler, and the core pipeline.



(a) Execution time

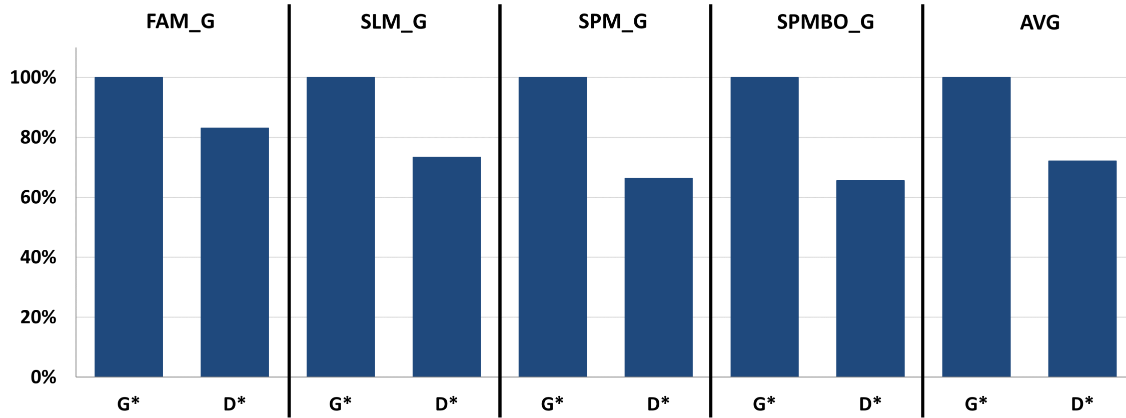


(b) Dynamic energy

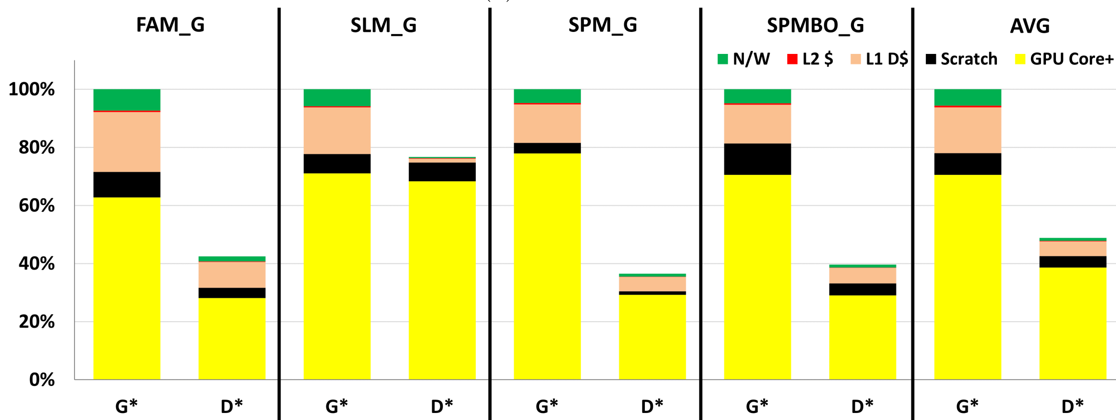


(c) Network traffic

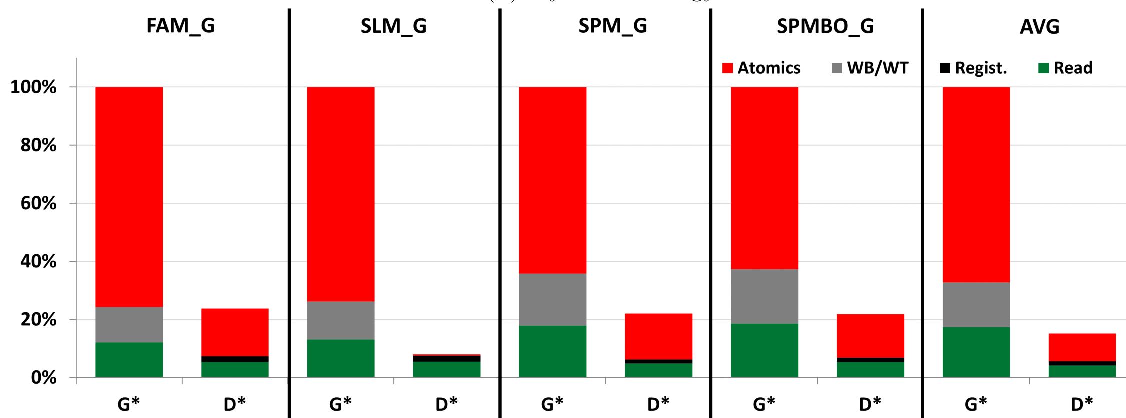
Figure 3.1: GPU coherence (G^*) and DeNovo (D^*) configurations with real GPU applications (no intra-kernel synchronization), normalized to G^*.



(a) Execution time

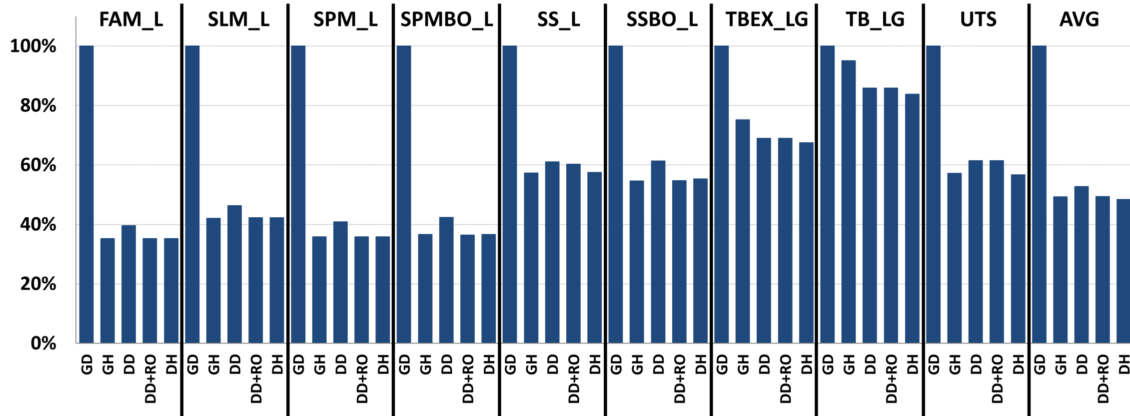


(b) Dynamic energy

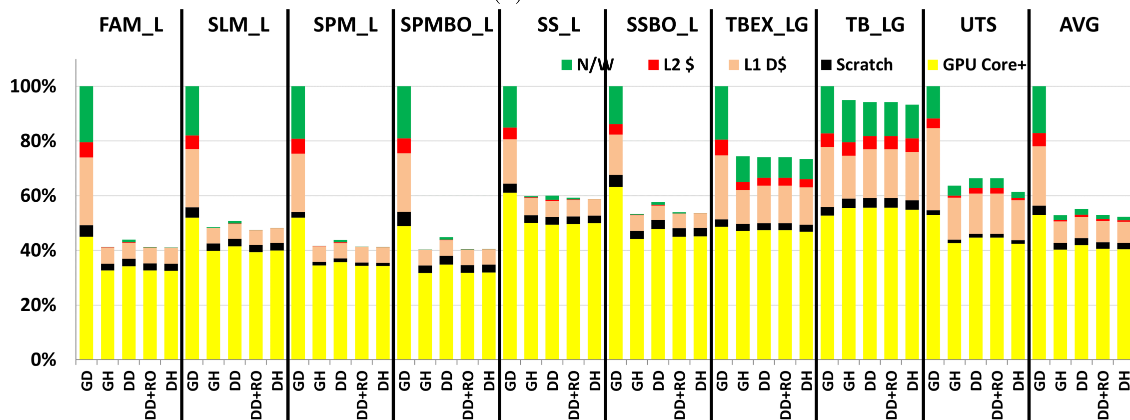


(c) Network traffic

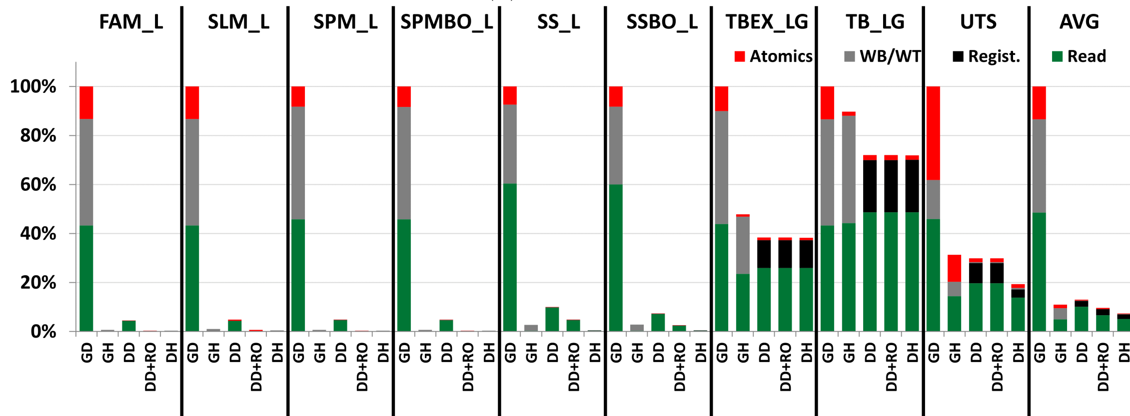
Figure 3.2: GPU coherence ($G*$) and DeNovo ($D*$) configurations with synchronization benchmarks that use only global synchronization, normalized to $G*$.



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 3.3: All configurations with synchronization benchmarks that use mostly local synchronization, normalized to *GD*.

Table 3.2: Benchmarks with input sizes. All thread blocks (TBs) in the synchronization microbenchmarks execute the critical section or barrier many times. Microbenchmarks with local and global scope are denoted with a '_L' and '_G', respectively.

Benchmark	Input
No Synchronization	
Backprop (BP)[39]	32 KB
Pathfinder (PF)[39]	10 x 100K matrix
LUD[39]	256x256 matrix
NW[39]	512x512 matrix
SGEMM[106]	medium
Stencil (ST)[106]	128x128x4, 4 iters
Hotspot (HS)[39]	512x512 matrix
NN[39]	171K records
SRAD v2 (SRAD)[39]	256x256 matrix
LavaMD (LAVA)[39]	2x2x2 matrix
Global Synchronization	
FA Mutex (FAM_G), Sleep Mutex (SLM_G), Spin Mutex (SPM_G), Spin Mutex+backoff (SPMBO_G),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Local or Hybrid Synchronization	
FA Mutex (FAM_L), Sleep Mutex (SLM_L), Spin Mutex (SPM_L), Spin Mutex+backoff (SPMBO_L), Tree Barr+local exch (TBEX_LG), Tree Barr (TB_LG),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Spin Sem (SS_L), Spin Sem+backoff (SSBO_L)[107]	3 TBs/CU, 100 iters/TB/kernel, readers: 10 Ld/thr/iter writers: 20 St/thr/iter
UTS[55]	16K nodes

Table 3.3: Simulated heterogeneous system parameters.

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
CUs	15
Memory Hierarchy Parameters	
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
Store Buffer Size	256 entries
L1 MSHRs	128 entries
L1 hit latency	1 cycle
Remote L1 hit latency	35−83 cycles
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

The overheads of DeNovo ownership have a negligible effect on performance for the real applications, which exhibit no intra-kernel synchronization. DeNovo does, however, reduce network traffic significantly for the Lava application, in which written data fits in the local cache but is

written through to the LLC when GPU coherence is used. For microbenchmarks with global synchronization, DeNovo is uniquely able to exploit reuse in owned data, and this benefit outweighs any ownership costs. As a result, DeNovo reduces execution time by on average 28%, energy by on average 51%, and network traffic by on average 81% relative to GPU coherence.

Figure 3.3 compares the execution time, energy, and network traffic of different coherence and consistency configurations for the microbenchmarks with local synchronization. *GD* is the worst configuration by far for all microbenchmarks due to the high synchronization costs. Performance can be improved significantly by moving to the HRF memory model, or by using DeNovo coherence. *GH* offers a slightly better improvement than DeNovo alone for most workloads. Relative to *GD*, *GH* reduces execution time, energy, and network traffic by on average 46%, 42%, and 89%, respectively. Relative to *DD*, *GH* reduces execution time, energy, and network traffic by 10%, 3%, and 16%. *GH* is able to outperform *DD* here because *DD* must still invalidate non-owned data at local synchronization points. The performance difference between HRF and DeNovo is eliminated (and slightly reversed) when DeNovo is enhanced with the read-only optimization (*DD+RO*), since this avoids invalidating read-only data at synchronization points. Unsurprisingly, *DH* performs best overall since it gets the benefits of both DeNovo ownership and scoped synchronization, reducing execution time, energy, and network traffic by 4%, 1%, and 33% relative to *GH*.

These results show that DeNovo has little effect on conventional applications with minimal synchronization, and is able to improve performance significantly relative to scope-free GPU coherence when synchronization is frequent. When synchronization is global, DeNovo is uniquely capable of improving the performance of conventional GPU coherence. When synchronization is local, DeNovo offers close to the same performance benefits as scoped synchronization (the gap is further reduced with the read-only optimization) without complicating the DRF memory model. When an HRF memory model is assumed, DeNovo+HRF outperforms GPU coherence+HRF.

3.3 Heterogeneous Lazy Release Consistency (hLRC+DRF)

While DeNovo is able to offer scalable and efficient GPU coherence by exploiting locality in written data across local and global synchronization, it does not completely avoid the overheads of flush and invalidation actions when synchronization is local. In addition, it adds some overheads to the

GPU coherence protocol by requiring the LLC to track ownership for all written data.

In this section we introduce heterogeneous lazy release consistency (hLRC) as an alternative coherence strategy, which is inspired in part by DeNovo ownership principles. Rather than exploiting locality in written data, hLRC exploits synchronization locality (similar to HRF and RSP) to limit synchronization costs. Like DeNovo, it is simple (uses a DRF consistency model), scalable (does not require broadcast operations), and naturally supports dynamic sharing.

3.3.1 Design

hLRC combines the principles of lazy release consistency, which has previously been used to reduce wasteful communication in distributed CPU shared memory systems [60], with insights from DeNovo and RSP. Like lazy release consistency for distributed CPUs, hLRC associates each atomic variable with the location it was last accessed. Coherence actions are then performed only when the location of the atomic variable changes (including when the variable is first brought into a cache), because this indicates a possible inter-core synchronization. Since this strategy does not require any scope information from software, hLRC is able to avoid the complexities of the HRF memory model.

In order to trigger the appropriate coherence actions when an atomic variable changes location, hLRC must track and serialize accesses to each atomic variable. This is accomplished by obtaining ownership for every atomic access. Ownership in hLRC is based on DeNovo ownership, described in Section 3.2. Since owned data is not affected by bulk coherence actions, a locally owned atomic variable will remain owned until ownership is requested by another core, or until the variable is evicted for space. While DeNovo uses registration for both atomic accesses and normal data stores, a key distinction of hLRC is that only atomic accesses require ownership. By only requesting ownership for atomic accesses, hLRC reduces both the amount of owned data that must be tracked at the L2 as well as the probe bandwidth and latency incurred by requests for remotely owned data.

Implementing Synchronization Semantics

Like RSP, hLRC is able to completely avoid bulk coherence actions when synchronization is local, and only performs flushes and invalidates when there is a potential for remote communication.

However, unlike RSP, hLRC does not rely on software-provided scopes to know when synchronization is local; it relies on the dynamic locality of the atomic synchronization variable itself. Every synchronization access is required to obtain ownership for the target atomic variable whether the access is a read, write or RMW. Whenever two threads on remote cores synchronize through an acquire-release access pair to an atomic variable, an ownership change will occur. When this is detected by the system, prior accesses on the releasing core (the producer synchronization access) must be made visible to subsequent accesses at the acquiring core (the consumer synchronization access). These actions are described next.

Release semantics: Release semantics in hLRC are not triggered at the time of the release synchronization access, but rather are delayed until a possible release-acquire pair with a remote core is detected. Since the release access will have obtained ownership and any subsequent acquire access must also obtain ownership, a remote synchronization pair is only possible when ownership changes. Therefore, whenever a cache loses ownership for an atomic variable, hLRC will flush any dirty data in the store buffer, making all prior accesses visible at the backing cache. If a cache loses ownership due to a remote access, this flush must complete *before* the atomic's state and value is updated in the requesting core. A cache may also lose ownership due to an eviction, in which case the flush must complete before the atomic's state and value are updated in the backing cache.

Acquire semantics: Acquire semantics in hLRC are triggered whenever a cache obtains ownership for an atomic variable. When a synchronization access misses in the local cache and must request the atomic variable from the backing cache or a remote owner, that synchronization access may be forming the acquire access in a remote acquire-release synchronization pair. To ensure the prior accesses on the release side are made visible to subsequent accesses on the acquire side, hLRC triggers a self-invalidation of any non-owned data *after* the cache obtains ownership for the atomic (but before any subsequent data accesses are executed).

In this way, hLRC avoids coherence actions for local synchronization while ensuring that any synchronizing pair of atomics executed in separate cores will be accompanied by a release flush action at the cache of the first atomic access (occurring after the memory access), and an acquire invalidation at the cache of the second atomic access (occurring before the memory access). This conservative triggering of coherence actions is sufficient to guarantee SC for DRF. However, it can also cause excessive coherence actions if atomic locality is low, evictions are frequent, or atomic

Table 3.4: Simulated heterogeneous system parameters.

128 GPU cores with below configuration	
Frequency	1 GHz
L1 data cache	16kB, 64B line, 16-way, 4 cycles, delivers one line every cycle
Memory Hierarchy	
L2 cache	4MB, 64B line, 16-way, 24 cycles write-through (write-back for Owned data)
1 instr. cache/4 GPU cores	32kB, 64B line, 8-way, 4 cycles
DRAM	DDR3, 32 channels, 500MHz

Table 3.5: Workloads and inputs.

Benchmark	Graph Inputs	Graph Sizes
Single Source Shortest Path (SSSP)	1: USA-road-d.BAY 2: USA-road-d.COL 3: c-68	7.86 MB 21.3 MB 4.27 MB
Graph Coloring (color)	1: ecology1 2: coAuthorsDBLP 3: dictionary28	35.9 MB 18 MB 1.64 MB
PageRank (PR)	1: USA-road-d.BAY 2: c-68 3: OPF_10000	7.86 MB 4.27 MB 3.57 MB

accesses do not have acquire-release semantics.

3.3.2 Evaluation

To evaluate hLRC, we simulate a CPU-GPU system using an extended version of the publicly available AMD gem5 APU simulator [13]. The simulated GPU contains 128 GPU cores in total. Each GPU core has four SIMD units and a private L1 data cache. All L1 data caches and instruction caches are connected to a unified L2 cache that is then connected to system memory through a memory controller shared by an on-chip CPU. These components are distributed evenly across an 8x16 mesh network. Table 3.4 describes the detailed parameters of the simulated system. Note that this evaluation uses a hierarchical simulation framework, which is different from the flat 2-level cache structure used for performance evaluation in Section 3.2.

By default, a write-through, write-allocate policy at the L1 and L2 is used for all data. An acquire operation triggers a single-cycle flash invalidation of the L1 cache, and a release operation triggers a flush of the L1 store buffer, which is implemented as a FIFO. To support DeNovo and hLRC we introduce an Owned state to the L1 and L2 caches. When using hLRC or DeNovo, L1 and L2 caches are write-back and write-allocate for owned data. hLRC only performs acquire and release actions when ownership changes, as described in Section 3.3.1. For both DeNovo and hLRC,

owned data is not invalidated on a self-invalidation or written back on flush.

We use three benchmarks from the Pannotia GPU graph analytics benchmark suite [38]: Single-source shortest path (**SSSP**), Graph coloring (**color**), and PageRank (**PR**). Each of these applications converges on a solution by iteratively processing all nodes in a graph. Processing a node involves visiting each of the node's neighbors, so load imbalance is introduced through variation in degree and locality.

Each application has been modified to use local task queues and work stealing. The input-dependent load imbalance of graph analytics workloads makes them a good fit for work stealing, which is a prime example of the dynamic sharing patterns enabled by RSP, DeNovo, and hLRC. In addition, the high expected (but statically unpredictable) synchronization locality of task queue access in a work stealing pattern makes it a good fit for hLRC.

Graph inputs are chosen from the Florida sparse matrix collection [48]. The inputs that we evaluated for each workload are listed in Table 3.5. These inputs were selected to fully utilize all 128 GPU cores.

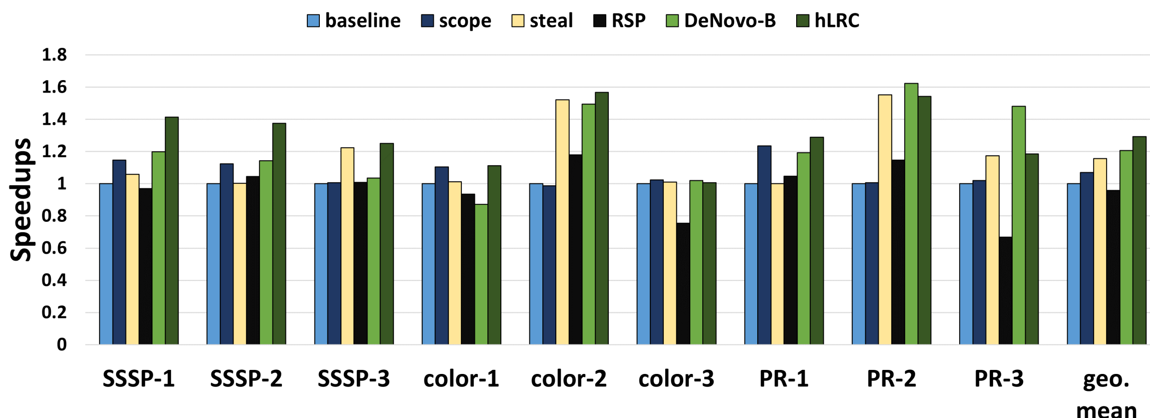
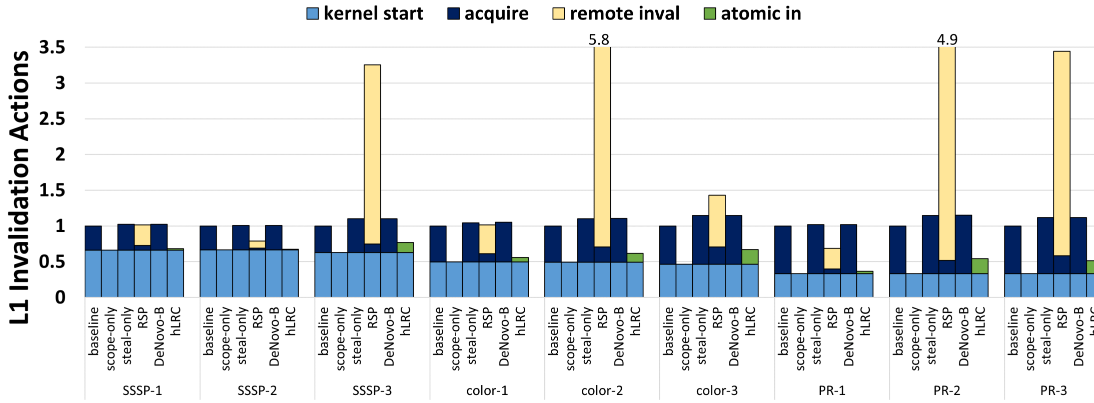


Figure 3.4: Performance (speedup relative to baseline).

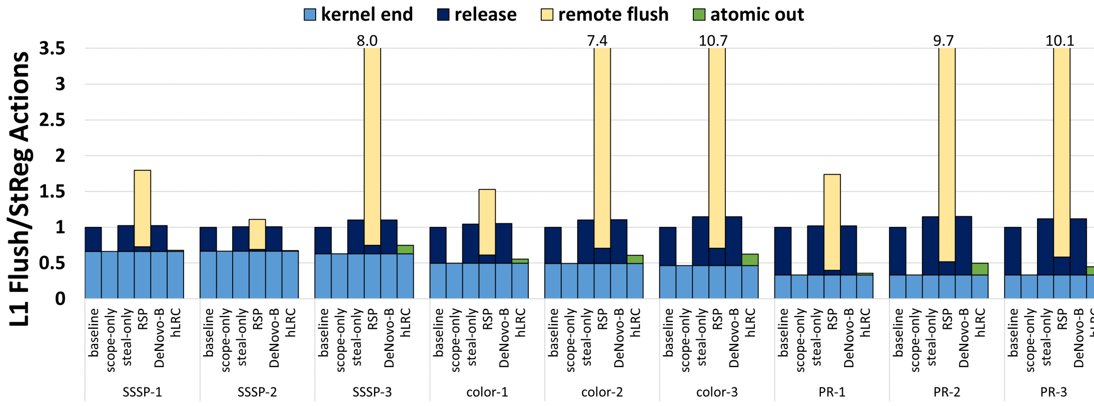
We measure the performance of 6 coherence and stealing configurations:

- **baseline**: Represents a conventional GPU coherence strategy where global synchronization is used to access local task queues only (work stealing is disabled).

- **scope**: Shows the individual benefit of local synchronization (locally scoped synchronization is used for accessing local task queues only and stealing is disabled).

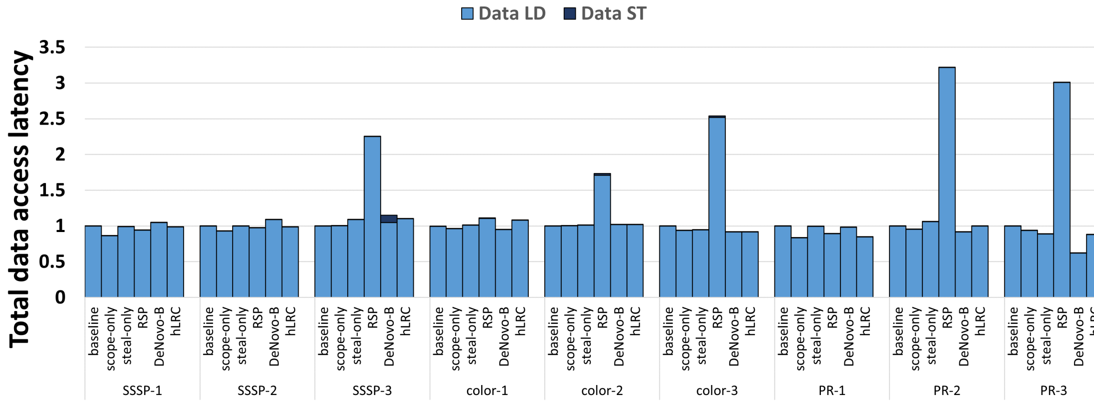


(a) Total invalidation coherence action counts, normalized to baseline

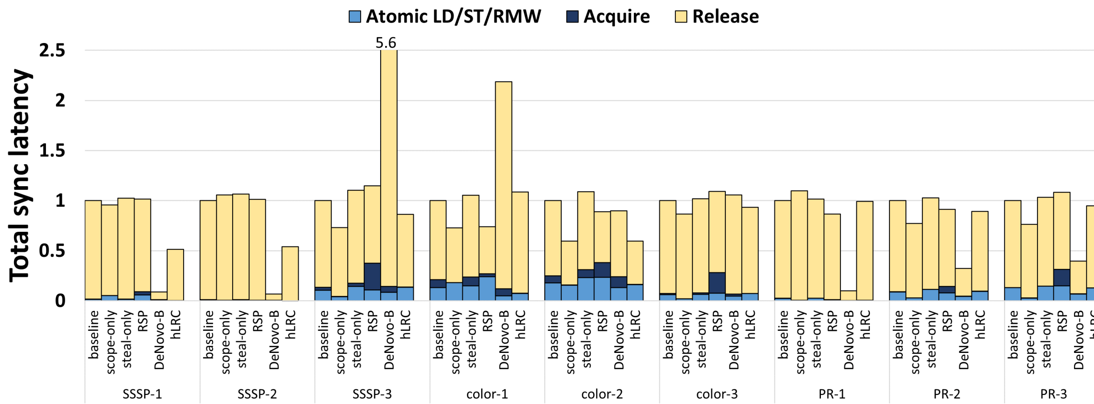


(b) Total flush coherence action counts, normalized to baseline

Figure 3.5: Breakdowns of invalidation and flush action types.



(a) Total latency for data loads and stores, normalized to baseline



(b) Total latency of all synchronization actions, normalized to baseline

Figure 3.6: Total latency breakdowns for data accesses and synchronization actions.

- **steal**: Shows the individual benefit of work stealing (task stealing is enabled and global synchronization is used for all queue accesses). Note that efficient dynamic sharing is not supported under conventional GPU coherence, so these benefits are mutually exclusive.

- **RSP**: Represents remote scope promotion, which uses locally scoped synchronization for accessing a local task queue, but requires RSP semantics and broadcast actions for attempting to steal from a remote task queue.

- **DeNovo-B**: Represents a simplified version of the DeNovo protocol that tracks ownership at block granularity (this differs from the implementation used in Section 3.2).

- **hLRC**: Represents heterogeneous lazy release consistency.

Both DeNovo-B and hLRC may steal from remote queues and do not need to specify scope semantics with their synchronization accesses.

Figure 3.4 compares the speedup for each configuration relative to baseline. On average, the scope-only configuration improves performance by 7% relative to baseline, the steal-only configuration improves performance by 16%, the RSP implementation causes a 4% decrease in performance, DeNovo-B improves performance by 21%, and hLRC improves performance by 29%.

Each configuration triggers invalidation and flush actions at different rates and for different reasons. To illustrate this, Figures 3.5a and 3.5b break down the invalidation and flush actions, respectively, based on cause. All configurations trigger invalidation and flush actions at the start and end of a kernel (labeled *kernel start* and *kernel end*). In the steal-only, RSP, and DeNovo-B configurations, accesses to the task queues involve acquire and release semantics which trigger local invalidations and flushes (labeled *kernel start* and *kernel end*). In RSP, invalidations and flushes may also be triggered by a remote cache when RSP semantics are used to steal. Finally, hLRC triggers invalidations and flushes based on atomic ownership changes rather than synchronization semantics. An invalidation is triggered when a cache obtains ownership for an atomic, and a flush is triggered when a cache loses ownership for an atomic (labeled *atomic in* and *atomic out*).

Figure 3.6a shows the combined latency of all non-atomic data accesses broken down by load and store operations and normalized to baseline. Figure 3.6b shows the combined latency of all acquire operations, release operations, and atomic accesses, normalized to baseline. This helps explain how

data access latency and atomic access latency (which are more numerous than data accesses but more likely to be on the critical path) are affected by the configurations studied. *Acquire* includes the latency of the *kernel start* and *acquire* actions in Figure 3.5a, *Release* includes the latency of the *kernel end* and *release* actions in Figure 3.5b, and *Atomic LD/ST/RMW* includes latency from the *atomic in* and *atomic out* invalidate/flush actions.

Using these detailed graphs, we next compare the performance of the RSP, DeNovo-B, and hLRC configurations.

RSP: As expected, RSP scales poorly to 128 cores. The broadcast lock, invalidate, and flush commands of the RSP implementation greatly increase coherence actions, data access latency, and synchronization latency. This ultimately degrades performance relative to the baseline configuration by up to 33%.

DeNovo-B: The DeNovo-B configuration scales well to 128 cores and can exploit significant cache locality in the presence of work stealing, delivering the best performance for multiple workloads. However, it does not provide the benefits of both scope-only and read-only for multiple other workloads. DeNovo-B differs from RSP and hLRC in that it obtains local registration for all stored data, and it performs coherence actions for every atomic. DeNovo-B's L2 inclusivity for all writes causes contention and evictions at the L2 cache, which in turn can increase release latency and fill up the store buffer, stalling subsequent access (evident in the release latency, acquire latency, and store latency in SSSP-3 and color-1). However, the L2 functions as a write-back cache for dirty data, enabling improved reuse of written data at the L2 (all other configurations, including hLRC, maintain consistency by invalidating L2 cache lines when writing through to memory). Thus, DeNovo-B is better able to exploit reuse at the L2 between GPU cores, which is evident in the decreased release latency and load latency for PR-2 and PR-3. For SSSP-1, SSSP-2, and PR-1, DeNovo-B suffers from frequent invalidations, which causes L1 read reuse to suffer and data access latency to increase slightly. However, by performing coherence actions at every acquire or release, DeNovo-B is less sensitive than RSP and hLRC to low synchronization locality (see color-3, PR-2, and PR-3).

hLRC: By automatically avoiding coherence actions when synchronization is local, hLRC is able to exploit the benefits of both work stealing and cache reuse. As Figures 3.5a and 3.5b show, hLRC decouples flush and invalidate operations from acquire and release accesses, reducing the

frequency of flush and invalidate operation when synchronization locality is high, which in turn reduces load, acquire, and release latency. However, since hLRC's benefits rely on synchronization locality, its gains are less pronounced when stealing is frequent, and it performs roughly the same as the steal-only configuration when steal-only is dominant (SSSP-3, color-2, PR-2, PR-3). This is because for every acquire-release pair between remote GPU cores, hLRC incurs the latency of serially executing flush and invalidate actions as part of the acquiring atomic access, rather than preemptively flushing at the release. Although the increase in atomic latency (relative to DeNovo) is in most cases outweighed by a decrease in release latency (Figure 3.6b), atomic access latency is more likely to be on the critical path, so it can have a larger impact on performance.

Overall, RSP, DeNovo-B, and hLRC all attempt to simultaneously optimize for cache locality and work stealing, but only hLRC is able to consistently match or exceed the performance of the best of scope-only and steal-only for the workloads studied. RSP's broadcast operations are unable to scale to 128 GPU cores, and result in an average 4% performance degradation. For most workloads, both DeNovo and hLRC are able achieve the benefits of steal and scoped synchronization, with an average 21% and 29% speedup relative to baseline, respectively. DeNovo excels when there is locality available in written data and suffers when the overheads of data ownership (increased L2 pressure and added delay for remote requests) are dominant, while hLRC excels when synchronization locality is high and suffers when synchronization locality is low. Overall, the work stealing workloads studied tend to exhibit more synchronization locality than locality in written data, and hLRC outperforms DeNovo-B by on average 7%.

3.4 Summary

To avoid the overheads of complex CPU coherence protocols, modern GPU caches use a coherence strategy designed for simplicity and scalability. They rely on DRF memory models and perform bulk invalidation and flush operations at synchronization points. This can be efficient for many conventional GPU applications, but performance suffers for emerging applications that exhibit fine-grained synchronization. Scoped synchronization can be used with the HRF memory model to skip coherence actions when synchronization is local, but it complicates the DRF memory model by requiring software to have static knowledge of the relative locations of synchronizing threads. This

added complexity complicates the job of programmers and compilers, and it precludes the use of scopes whenever communication patterns are irregular or unpredictable.

Both DeNovo and hLRC enable efficient coherence in the presence of frequent synchronization without the complexity of scopes. DeNovo reduces the cost of synchronization actions by obtaining ownership for writes and atomic accesses, enabling improved reuse in this data. This allows it to reduce execution time, energy, and network traffic for applications that use global synchronization, and to approach the performance of GPU coherence with scopes in applications where synchronization is statically known to be local. In contrast, hLRC reduces the frequency of flush and invalidation actions by exploiting locality in synchronization accesses. It combines insights from DeNovo, RSP, and lazy release consistency for CPUs to track atomic locality and only perform flush and invalidate actions when potential synchronization between remote cores is detected. This makes it particularly well suited for work stealing applications, which tend to exhibit frequent local synchronization and dynamic sharing.

Both coherence innovations have disadvantages. DeNovo only exploits reuse in owned data, and ownership requires extra state bits and L2 inclusivity. If locality in written data and atomics is low, obtaining ownership can be wasteful and incur added latency for remote requestors. hLRC triggers flush and invalidate coherence actions at atomic ownership changes rather than synchronization accesses, which can lead to excessive coherence actions or move delay to the critical path if synchronization locality is low. However, in general these costs seem to have a minimal performance impact. In our evaluations, DeNovo and hLRC consistently match or exceed the performance of the baseline configurations for the workloads studied, both in microbenchmarks and real GPU applications.

In addition, many of these disadvantages can be mitigated using optimizations that exploit information from software. DeNovo's primary disadvantage is its inability to exploit reuse in non-owned data. DeNovo regions, originally described by Choi et al. [42], enable selective invalidation of non-owned data at synchronization points. This allows improved cache reuse for valid data as well as owned data (the read-only region optimization from Section 3.2.3 is one example of this technique). hLRC's primary disadvantage is the possibility of excessive serialized flush and invalidation actions that can be triggered when synchronization locality is low. Scope semantics are unnecessary for functionality in hLRC, but they can be used to avoid excessive flush and invalidation actions

by communicating expected synchronization locality. For example, if synchronization locality is expected to be low, global scope may be used for an access, and the hLRC system can choose to not obtain ownership for that access, instead preemptively performing the cache flush or invalidation associated with the synchronization access. When synchronization locality is low, then, scopes can be used to reduce ownership transfer, improving hLRC performance for a broader range of applications.

Even without these optimizations, both DeNovo and hLRC show that efficient GPU coherence is possible without forcing the programming to deal with added memory model complexity. Both techniques offer advantages over GPU coherence, but which coherence strategy is preferable will depend on the locality properties of the target workload.

Chapter 4

DRF-Relaxed: Relaxed Atomics with SC-Centric Semantics

In modern GPUs, synchronization costs can quickly form a performance bottleneck for workloads with frequent synchronization. Scoped synchronization and relaxed atomics are two common ways to reduce the cost of synchronization. Chapter 3 discussed the complexities of scoped synchronization and proposed changes to the coherence protocol to improve synchronization efficiency. In this chapter we describe the benefits and limitations of relaxed atomics as currently defined, and we propose DRF-Relaxed (or DRFrlx), a consistency model that enable us to achieve the performance benefits of atomic relaxation while retaining simple SC-centric semantics.

4.1 Background: Relaxed Atomics

In a DRF compliant system, synchronization accesses enforce ordering between memory accesses in a thread. This enforcement typically involves waiting for all prior memory accesses to complete before issuing an atomic request and waiting for a pending atomic to complete before issuing any subsequent memory accesses. In systems that use self-invalidation or write-through caches (e.g., GPUs) it may also involve performing a flush of dirty data and a bulk cache invalidation before and after the atomic access. Atomic semantics that adhere to these constraints are called SC atomics, or paired atomics. They may have a negative impact on performance, but they are required to ensure sequentially consistent (SC) behavior in modern DRF (or HRF) GPU memory models. Such SC-centric memory models are popular because SC behavior is easy to reason about.

In some cases it is possible for the system to disregard these constraints and still provide acceptable (in some cases non-SC) behavior. This has motivated the addition of *relaxed atomics* to memory models for C++ and HSA, among others. By relaxing ordering constraints, relaxed atomics can improve performance at the cost of SC guarantees. Performance gains are especially

impressive in GPU systems, where expensive bulk coherence actions are required to enforce ordering constraints. Figure 4.1 shows the speedup from using relaxed atomics rather than SC atomics for a set of GPU benchmarks, each run with multiple inputs. The benchmarks were selected for their atomic access frequency and were executed on an NVIDIA GeForce GTX680. Although not all applications benefit from atomic relaxation, many applications experience significant speedups (up to 99x for PageRank).

Unfortunately, the introduction of relaxed atomics greatly complicates program semantics, and not only because it breaks guarantees of SC behavior. Formalizing any kind of sensible program behavior in the presence of relaxed atomics has proven difficult due to the problem of "out-of-thin-air" values. Out-of-thin-air values refer to arbitrary values that can arise in an execution due to self-satisfying speculative memory operations that are part of a cyclic dependency. This type of behavior makes it extremely difficult to reason about the values that could be returned by memory accesses and should be disallowed, but prohibiting this without simultaneously prohibiting some desirable hardware optimizations remains an unsolved formalization challenge [18, 28, 6, 19, 29, 93, 58].

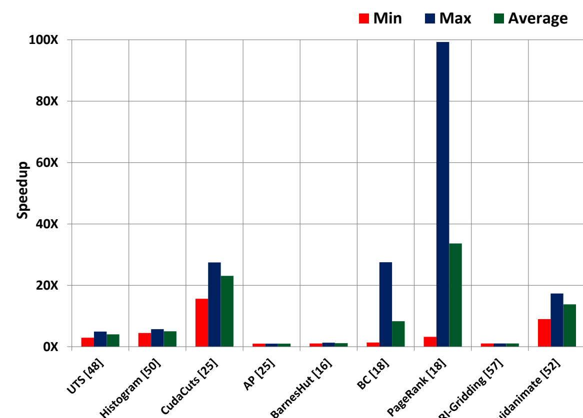


Figure 4.1: Relaxed atomics speedup on a discrete GPU.

There have been many attempts to formalize relaxed atomics (described in more detail in Section 7.3), but at this point all are acknowledged to have significant limitations. Furthermore, all require giving up the familiar interface of SC, even if there is a single relaxed atomic in the program, including one buried in invisible library code. There is clearly a need for a formalism that enables the huge performance gains of relaxed atomics without resorting to complicated or poorly defined semantics.

4.2 Relaxed Atomics Use Cases

The DRFrlx memory model is based on a different goal from that of prior attempts at formalism. Offering clear, simple semantics for every possible use case of relaxed atomics has proven to be

extremely difficult, if not impossible. Instead, we draw from the intuition behind the widely adopted DRF consistency model: look at how relaxed atomics are used in practice, and only worry about offering semantics for the types of applications that can practically benefit from atomic relaxation. Thus, in developing DRFrlx we identified situations where relaxed atomics are useful and safe, we formalized the properties of safe usage, and we defined how DRFrlx can offer SC-centric semantics to a program that uses relaxed atomics under these conditions.

After reaching out to developers, vendors, and researchers active in this area, we identified 5 common useful and safe relaxed atomic categories. Sections 4.2.1-4.2.4 informally describe these new relaxed atomic types, the use cases that motivate them, and the conditions for safe use. Much of the terminology used in these sections is formally defined in Section 4.3, which formally specifies the DRFrlx memory model.

4.2.1 Unpaired Atomics

The first atomic type, unpaired atomics, has already been formalized in the DRF1 memory model [3]. They are motivated by the use case where an atomic access is not being used to form a paired chain between racy data accesses. One common example of this is the **work queues** access pattern illustrated in code listing 4.1. A worker thread periodically checks the occupancy of a work queue. If the queue is empty, no further action is taken on the queue until the next check. If there is work to be done, the worker will access some of the data in the queue. SC atomics are only needed to protect data accesses when the queue is not empty. The occupancy check may be relaxed with respect to data accesses because any subsequent queue accesses will be protected by a separate SC atomic access.

DRF1 extends the DRF0 memory model [4], which does not distinguish different atomic types and considers all atomics SC atomics. Unpaired atomics may not be reordered with other SC and unpaired atomics, but can be reordered with data accesses, which means they may elide flush and invalidate actions in a GPU coherence protocol. As a result, they may be used to avoid a data race on the memory accesses they are used for, but they cannot be used as part of a paired synchronization to order racy data accesses in an SC for DRF memory model. This is formalized in their exclusion from the definitions of *synchronization order* and *happens-before* in Section 4.3.

```
struct Task;
struct MsgQueue {
  atomic<int> _occupancy = 0;

  Task * dequeue() {
    if (_occupancy.atomic_load(mem_order_seq_cst) == 0) {
      return NULL;
    } else { ... }
  }
  int occupancy() {
    return _occupancy.atomic_load(mem_order_relaxed);
  }
  ...
} globalQueue;

// Thread t1 (service thread):
  void periodicCheck() {
    if (globalQueue.occupancy() > 0) {
      Task * t = globalQueue.dequeue();
      if (t != NULL)
        t.execute();
    }
  }
```

Listing 4.1: Work Queue example [52].

Commutative Atomics

Commutative atomics are motivated by the use case where there are multiple associative and commutative updates being concurrently performed on some data such that the order in which the updates are performed does not affect the final result of the program. A common example of this pattern is the **event counters** use case illustrated in code listing 4.2. In this code, multiple threads concurrently update a histogram of event counts based on the values read from a data array. If the return value of the atomic (the intermediate value at the time the update is performed) is never used by the updater and the final value for each bin is only read after some SC synchronization, the commutative updates can be safely reordered and still result in the same final value for each histogram bin.

Used correctly, commutative atomics may be reordered with both data accesses and other relaxed atomic accesses. The conditions for safe use require that, in any SC execution of program, the intermediate return value from any commutative atomic update is not used in the updating thread, and all commutative atomic updates only race with atomic accesses that are pairwise commutative and associative (e.g., atomic add and atomic subtract operations may race). If a commutative atomic is involved in a race that does not satisfy the above conditions, this constitutes a *commutative race*, formalized in Section 4.3.

```
atomic<int> count[NUM_BINS]; // all bins initialized to 0

// Threads 1..N:
threadNum = ...
chunkSize = ...
baseLoc = (threadNum * chunkSize);
...
for (i = 0; i < chunkSize; ++i) {
  binNum = data[baseLoc + i] % NUM_BINS;
  count[binNum].atomic_inc(mem_order_relaxed);
}
...

// Main Thread:
main() {
  launch_workers(); // launch worker threads
  ...
  join_workers();
  for (i = 0; i < NUM_BINS; ++i) {
    int r1 = count[i].atomic_load(mem_order_relaxed);
    ... // uses r1
  }
}
```

Listing 4.2: Event counters example [28, 33, 112, 94].

4.2.2 Non-Ordering Atomics

Non-ordering atomics are motivated by codes where synchronization accesses do not order any other data or synchronization accesses in the program. This essentially means that reordering the accesses with other data or relaxed atomic accesses in the thread will still result in SC behavior. An example of this pattern is given by the **flags** use case shown in code listing 4.3. In this example, worker threads repeatedly perform some computation, which may involve setting a dirty flag until they read a stop flag that has been set by a master thread. Relaxation is safe here because the atomic accesses to stop and dirty may be reordered with one another and the final result will always be consistent with an SC execution.

Non-ordering atomics may be reordered with data accesses and other relaxed atomics. The conditions for safe use require that every pair of racy atomic accesses in any SC execution of the program are not exclusively ordered (through program order and conflict order) by a path that includes non-ordering atomics. Such a race constitutes a *non-ordering race*, formally defined in Section 4.3.

```
atomic<bool> dirty = false, stop = false;

// Threads 1..N:
...
while (!stop.atomic_load(mem_order_relaxed)) {
  if (...) {
    dirty.atomic_store(true, mem_order_relaxed);
    ...
  }
  ...
}

// Main Thread:
main() {
  launch_workers(); // launch threads 1..N
  ...
  stop.atomic_store(true, mem_order_relaxed);
  join_workers();
  if (dirty.atomic_load(mem_order_relaxed))
    cleanup_dirty_stuff();
}
```

Listing 4.3: Flags example [112].

4.2.3 Speculative Atomics

Sometimes a memory access may be performed speculatively but its result will only be used if it is found to not have raced with any other concurrent accesses. Speculative atomics are designed for this scenario. An example of this pattern is shown in the **seqlock** use case illustrated in code listing 4.4. Seqlock enforces a single writer/multiple reader invariant in some shared data by surrounding read accesses to the data with SC atomics to check whether a concurrent write has executed concurrently with the read operation. If a write action is found to have raced with a read action, the read values are discarded and the read operation is retried. If the speculative reads in the reader operation are reordered and are performed concurrently with writes, these reads can return non-SC values. However, since the race is guaranteed to be detected and these non-SC values therefore are guaranteed to not be used, the final result will still be consistent with an SC execution.

Speculative atomics may be reordered with data accesses and other relaxed atomics. This relaxed atomic type can be safely used in a program as long as, in every SC execution of the program, any race involving speculative atomics involves at least one load (not a read-modify-write), and the return value of that load is not used. Any race involving speculative atomics that does not satisfy these requirements forms a *speculative race*, defined formally in Section 4.3.

```
atomic<unsigned> seq;
atomic <int> data1, data2;

T reader() {
  int r1, r2;
  unsigned seq0, seq1;
  do {
    seq0 = seq.atomic_load(mem_order_seq_cst);
    r1 = data1.atomic_load(mem_order_relaxed);
    r2 = data2.atomic_load(mem_order_relaxed);
    seq1 = seq.atomic_fetch_add(0, mem_order_seq_cst);
  } while ((seq0 != seq1) || (seq0 & 1));
  // uses r1 and r2
}

void writer(...) {
  unsigned seq0 = seq.atomic_load(mem_order_seq_cst);
  while ((seq0 & 1) || !seq.cmp_exchange(seq0, seq0+1, mem_order_relaxed)) {
    seq0 = seq.atomic_load(mem_order_seq_cst);
  }
  data1.atomic_store(..., mem_order_relaxed);
  data2.atomic_store(..., mem_order_relaxed);
  seq.atomic_store(seq0 + 2, mem_order_seq_cst);
}
```

Listing 4.4: Seqlocks example [25].

4.2.4 Quantum Atomics

The final relaxed atomic type is Quantum Atomics. While prior relaxed atomic types preserve SC semantics, quantum atomics are designed for codes where relaxation can lead to non-SC behavior, but this non-SC behavior is tolerable. On example of this is the **split counter** use case shown in code listing 4.5. In this example, a counter is split into multiple sub-counters, which may be concurrently read and updated by parallel threads. If atomic accesses to these sub-counters are relaxed, the values returned for partial sum reads may not be consistent with an SC execution. However, this non-SC behavior is tolerable for many applications as long as the loaded values are roughly representative of the current state of each sub-counter.

Quantum atomics may be reordered with data accesses and other atomic accesses. The issue of non-SC behavior and arbitrary out-of-thin-air return values is addressed by the introduction of a *quantum-equivalent program.* A quantum-equivalent program replaces every quantum load in a DRFrlx program with a call to a dummy random number generator. This quantum-equivalent program is then used to enumerate all possible SC executions when identifying illegal races. In addition to races defined above, a race between a quantum atomic store and a non-quantum access of any kind in a SC execution of the quantum-equivalent program constitutes a *quantum race*,

```
atomic<unsigned long> myCount[NUM_THREADS];
add_split_counter(v, tID) {
  val = myCount[tID].atomic_load(mem_order_relaxed);
  newVal = val + v;
  myCount[tID].atomic_store(newVal, mem_order_relaxed);
}
read_split_counter(tID) {
  sum = 0;
  for (i = 0; i < NUM_THREADS; ++i) {
    loc = ((tID + i) % NUM_THREADS);
    sum += myCount[loc].atomic_load(mem_order_relaxed);
  }
  return sum;
}

add_split_counter(1, 0); // Thread 0
r1 = read_split_counter(1); // Thread 1
add_split_counter(2, 2); // Thread 2
r2 = read_split_counter(3); // Thread 3
```

Listing 4.5: Split counters example [84].

formally defined in Section 4.3. This quantum-equivalence check forces the programmer to consider the unintentional side effects that can arise from non-SC behavior, including out-of-thin-air values.

The execution of a DRFrlx program on a DRFrlx compliant system must be consistent with an SC execution of the quantum-equivalent program, and must also preserve per-location SC and happens-before consistency in quantum atomics. Thus, quantum atomic semantics isolate the non-SC effects of this relaxed atomic type while still allowing programmers to think about possible executions and races in a SC-centric manner.

4.3 DRF-Relaxed Formal Definitions

The DRFrlx memory consistency model extends the DRF1 memory model defined by Adve and Hill, and much of the terminology used in the following definitions is based on this work [3]. An *operation* is a memory access that either reads a memory location (a load) or modifies a memory location (a store). Two memory operations *conflict* if they access the same memory location and at least one of them is a store. Two memory operations, *op1* and *op2*, are ordered by *program order* ($op1 \xrightarrow{\textbf{po}} op2$) if and only if both are from the same thread and *op1* is ordered before *op2* by the program text. An execution is *sequentially consistent (SC)* if there exists a total order, T, on all its memory operations such that (i) T is consistent with *program order* and (ii) a load L returns

the value of the last store S to the same location ordered before L by T. We refer to T as an **SC total order** for the execution. [1]

As with DRF1, all memory operations in a DRFrlx program are distinguished to the system as either data or atomic. In addition to the paired and unpaired atomic types defined in DRF1, [2] an atomic operation in DRFrlx may also be distinguished as *commutative, non-ordering, speculative, or quantum.*

4.3.1 Definitions for an SC Execution with SC total order T

Synchronization Order ($\xrightarrow{\textbf{so}}$): Let **X** and **Y** be two memory operations in an execution. **X** $\xrightarrow{\textbf{so1}}$ **Y** if and only if **X** and **Y** conflict, **X** is a paired synchronization write, **Y** is a paired synchronization read, and **X** is ordered before **Y** in the **SC total order**.

Happens-before ($\xrightarrow{\textbf{hb}}$): The happens-before relation is defined on the memory operations of an execution as the irreflexive transitive closure of po and so: $hb = (po \cup so)+$.

Race: Two operations **X** and **Y** in an execution form a race (under DRF1) if and only if **X** and **Y** are from different threads, they conflict with each other, and they are not ordered by happens-before.

Data Race: Two operations **X** and **Y** form a data race (under DRF1) if and only if they form a race and at least one of them is distinguished as a data operation.

Result of an execution: The memory state at the end of the execution.

Commutativity: Two stores or RMWs to a single memory location M are *commutative with respect to each other* if they can be performed in any order and yield the same value for M.

Commutative Race: Two operations **X** and **Y** form a commutative race if and only if:

- **X** and **Y** form a race,
- at least one of **X** or **Y** is distinguished as a commutative operation, and
- **X** and **Y** are not commutative with respect to each other or the value loaded by either operation is used by another instruction in its thread.

[1]For brevity, we refer the reader to [5] for formal definitions of several intuitive notions. Informally, an *execution* must obey correctness requirements for a single core. To accommodate *read-modify-writes* (RMW), the read (load) and write (store) of a RMW must appear together in an SC total order.

[2]Paired atomics are the equivalent of SC atomics in the C and C++ models [28].

Conflict Order ($\xrightarrow{\textbf{co}}$): **X** $\xrightarrow{\textbf{co}}$ **Y** if and only if **X** and **Y** conflict and **X** is ordered before **Y** in T.

Program/Conflict Graph and a Path [5]: The **program/conflict graph** of an execution is a directed graph where the vertices are the (dynamic) operations of the execution and the edges represent program order and conflict order. Below all paths refer to paths in the program/conflict graph.

Ordering Path [5]: A path from **X** to **Y** is called an **ordering path** if it has at least one program order edge and **X** and **Y** conflict.

Race Path: A **race path** from operation **X** to operation **Y** exists if and only if **X** $\xrightarrow{\textbf{co}}$ **Y** and there is no path from X to Y in the **program/conflict graph** that contains at least one **program order** edge.

Connect: For two conflicting atomics **X** and **Y**, **X connects Y** if there is a path between them in the **program/conflict graph** and either:

- the path has at least one **program order** edge, or
- there is a **race path** between **X** and **Y**

Non-ordering Race: Two operations, **X** and **Y** form a non-ordering race if and only if:

- **X** and **Y** form a race,
- both **X** and **Y** are distinguished as atomics and at least one of them is distinguished as a non-ordering atomic, and
- there exist racing (non-commutative) atomic accesses, **A** and **B** in at least one SC execution, such that **X** $\xrightarrow{\textbf{co}}$ **Y** is part of **A connects B** and there is no alternate way to connect **A** and **B** without passing through a non-ordering atomic access.

Speculative Race: Two operations, **X** and **Y**, form a speculative race if and only if they form a race, at least one of **X** or **Y** is distinguished as a speculative atomic, and either:

- both operations are stores, or
- the result of the load is observed by another instruction in the execution (i.e., the returned value is used by another instruction in the thread).

Quantum-Equivalent Program: We generate a quantum-equivalent program Pq from a program P as follows. Each quantum atomic load *ri = Y;* in P is replaced with a call to a conceptual random function *ri = random();* in Pq. Similarly, each quantum atomic store $Y = rj$ is replaced with a quantum store of a random value *Y = random().* A quantum RMW's load and store are both replaced as above.

Quantum Race: Two operations, **X** and **Y** form a quantum race if and only if they form a race, either **X** or **Y** is a quantum atomic store, and the other is not a quantum atomic.

Happens-Before Consistency: A load **L** must always return the value of a store **S** to the same memory location M in the execution. It must not be the case that $\mathbf{L} \xrightarrow{\mathbf{hb1}} \mathbf{S}$ or that there exists another store **S'** to M such that $\mathbf{S} \xrightarrow{\mathbf{hb1}} \mathbf{S'} \xrightarrow{\mathbf{hb1}} \mathbf{L}$.

Per-Location SC: There is a total order, T_{loc}, on all operations to a given memory location such that T_{loc} is consistent with happens-before-1, and that a load **L** returns the value of the last store to this location before **L** by T_{loc}.

4.3.2 DRF-Relaxed Model Definition

This section defines the DRFrlx consistency model based on terminology defined in the above sections.

DRF-Relaxed Program: A program is DRFrlx if and only if for every SC execution of the *quantum-equivalent* program, all memory operations can be distinguished by the system as either data, paired atomic, unpaired atomic, *commutative atomic*, *non-ordering atomic*, *speculative atomic*, or *quantum atomic*, and there are no data races, *commutative races*, *non-ordering races*, *speculative races*, or *quantum races* in the execution.

DRF-Relaxed Model: A system obeys the DRFrlx memory model if and only if the result of every execution of a DRFrlx program on the system is the result of an SC execution of the *quantum-equivalent* program. Additionally, every execution must obey happens-before consistency and per-location SC for all atomic accesses in the original program.

DRF-Relaxed Correctness Theorem: Theorem 1 describes a system with properties that are sufficient to correctly implement DRFrlx. The system used in the evaluation conforms to these properties. Although a proof is omitted for space, it follows the basic structure of DRF proofs in

prior work [5].

Theorem 1. *Assume a heterogeneous system is DRF1 compliant in that it allows reordering of data accesses with other data accesses and with unpaired atomic accesses, but enforces happens-before consistency and prevents reordering of paired and unpaired atomics. Assume also that it enforces per-location SC for all atomics and constrains DRFrlx's commutative, non-ordering, speculative, and quantum atomic operation completion/propagation in the same way as data operations. Such a system is DRFrlx compliant.*

4.4 Formalization in Herd

We also formalize DRFrlx in Herd [9], a tool for formalizing memory models in terms of allowed relations between memory accesses in different threads. Given a model definition and a program, Herd produces all possible executions of the program as constrained by the model, and flags any relations of interest as specified by the model (e.g., race conditions). Herd verification of DRFrlx consists of two models: a programmer-centric model and a system-centric model. The programmer-centric model (shown in code listing 4.6) defines and identifies illegal races under DRFrlx. The system-centric model (shown in code listing 4.7) generates all possible executions of a program that would be possible in an example hardware system that enforces DRFrlx constraints.

Numerous litmus tests were used to test these models. They include the use cases in Section 4.2, incorrectly labeled versions of these use cases, and various other tests designed to stress various racy and non-racy patterns. Although detailed results are omitted, for all litmus tests, the programmer-centric model correctly identifies races in the SC execution, and the system-centric model can only produce non-SC executions when the model allows it (i.e., when there is an illegal race or when quantum atomics are used).

4.4.1 Relaxed Atomics Herd Formalization

```
let at-least-one a = a*_ | _*a


let PairedR = (Paired & R)
let PairedW = (Paired & W)
```

```
let so1 = (PairedW * PairedR) & (rf | fr | co)+
let hb1 = (po | so1)+
let conflict = at-least-one W & loc
let race = (conflict & ext & ~(hb1 | hb1^-1)) \ (IW*_)
let data-race = race & (at-least-one Data)

(* comm-pair relates any two memory operations that are pairwise commutative (we omit the
    precise definition for space) *)
(* commutative race: a race involving a commutative access where either a) the accesses are
    not pairwise commutative *)
let comm-race1 = (race & (at-least-one Comm)) \ comm-pair
(* or b) the return value of an operation is observable *)
let comm-race2 = (race & (at-least-one Comm)) ; (addr | data | ctrl)
let comm-race = comm-race1 | comm-race2

(* pco: program-conflict order *)
(* pcoPO: pco that contains a po edge *)
let pco = (po | co | rf | fr)+
let pcoPO = po | (po ; pco) | (pco ; po ; pco) | (pco ; po)

(* opath-aloNO: ordering path with at least one NO atomic *)
let aloNO = (at-least-one NonOrder)
let pcoPO-NO-pco = (pcoPO & aloNO) ; pco
let pco-NO-pcoPO = pco ; (pcoPO & aloNO)
let pcoPO-aloNO = (pcoPO & aloNO) | pcoPO-NO-pco | pco-NO-pcoPO
let opath-aloNO = pcoPO-aloNO & conflict

(* valid ordering path 1: accesses to the same address *)
let valid-pco1 = ((po | co | rf | fr) & loc)+
let valid-po1 = po & loc
let valid-pcoPO1 = valid-po1 | (valid-po1 ; valid-pco1) | (valid-pco1 ; valid-po1 ; valid-
    pco1) | (valid-pco1 ; valid-po1)
let valid-opath1 = valid-pcoPO1 & conflict

(* valid ordering path 2: Unpaired/Paired accesses *)
let valid-pco2 = ((po | co | rf | fr) & (Paired | Unpaired)*(Paired | Unpaired))+
let valid-po2 = po & (Paired | Unpaired)*(Paired | Unpaired)
let valid-pcoPO2 = valid-po2 | (valid-po2 ; valid-pco2) | (valid-pco2 ; valid-po2 ; valid-
    pco2) | (valid-pco2 ; valid-po2)
let valid-opath2 = valid-pcoPO2 & conflict

(* non-ordering race: there is an ordering path between two accesses that contains a
```

```
    NonOrdering edge, and there are no alternate valid ordering paths *)
(* note: for simpler herd construction, this relation is defined between the accesses at the
     ends of the ordering path *)
let non-order-race = ((race \ data-race \ comm-race) & opath-aloNO) \ valid-opath1 \ valid-
    opath2

(* quantum race: Quantum races with non-quantum *)
let quantum-race = (race & (at-least-one Quantum)) \ (Quantum * Quantum)

(* speculative race: a race involving a speculative access where either a) both accesses are
     writes *)
let speculative-race1 = (race & (at-least-one Spec) & (W * W))
(* ... or b) the racy load is observable *)
let speculative-race2 = (race & (at-least-one Spec)) ; (addr | data | ctrl)
let speculative-race = speculative-race1 | speculative-race2

let illegal-race = data-race | comm-race | non-order-race | quantum-race | speculative-race

(* limit to SC executions *)
acyclic (po | rf | co | fr)
(* RMWs to happen atomically *)
empty rmw & (fre ; coe)

(* Identify any races in SC executions *)
flag ~empty (illegal-race) as IllegalRace
```

Listing 4.6: DRFrlx's programmer-centric model in Herd: defining and identifying illegal races in a program.

4.5 Evaluation

This evaluation measures the benefits of different levels of atomic relaxation in emerging GPU applications as well as how these benefits are affected by the choice of coherence strategy. DRF0 is used as the baseline consistency model configuration and forces all atomic accesses to use paired atomic semantics, requiring a cache flush and invalidate at every atomic write and read, respectively. DRF1 relaxes ordering constraints between unpaired atomics and data accesses, showing the benefit of avoiding cache flush and invalidation actions at each such atomic access. Finally, DRFrlx relaxes ordering constraints between relaxed atomic types, showing the benefits of overlapping atomic

```
let at-least-one a = a*_ | _*a

(* atom-pair relates any two atomic accesses (relaxed or not) *)
let per-loc-atomic = loc & (atom-pair) & (po | co | rf | fr)

(* Which po orderings are enforced in example system: accesses to the same address,
    successive unpaired/paired accesses, acq/rel fences*)
let inst-order = po & (loc | ((Acq | Rel | Unpaired) * (Acq | Rel | Unpaired)) | _*Rel | Acq
    *_)
(* control/data dependence requirement - don't speculatively issue stores *)
let control-order = (data | ctrl | addr)+

(* enforce per-location sc for atomics *)
acyclic(per-loc-atomic)
(* make sure temporally enforced relations are acyclic *)
acyclic(inst-order | rf | fr | control-order)
(* force RMWs to happen atomically *)
empty rmw & (fre;coe)

(* Identify any non-sc behavior *)
flag ~acyclic (po | rf | co | fr) as NonSC
```

Listing 4.7: DRFrlx's system-centric model: Used to detect possible non-SC behavior in an example DRFrlx system.

accesses in the memory system. The two coherence strategies studied are GPU coherence and DeNovo, described in Chapter 3. The main difference between the two is that DeNovo obtains ownership for writes and atomic accesses, while GPU coherence writes through dirty data and performs atomic accesses at the L2.

We evaluate the following 6 configurations which include the full set of The full set of configurations and associated abbreviations are as follows:

- **GD0**: GPU coherence with DRF0 consistency. This is the baseline configuration.
- **GD1**: GPU coherence with DRF1 consistency.
- **GDR**: GPU coherence with DRFrlx consistency.)
- **DD0**: DeNovo with DRF0 consistency.
- **DD1**: DeNovo with DRF1 consistency.
- **DDR**: DeNovo with DRFrlx consistency.

The GPU applications studied include both synthetic microbenchmarks and real benchmark applications. The microbenchmarks represent the use cases described in Section 4.2. The benchmarks are a set of emerging GPU applications that were selected for having frequent synchronization

that may be safely relaxed. The evaluation is performed using the same integrated architectural CPU-GPU simulator used in Section 3.2.3 but with added support for relaxed atomics.

Figures 4.2a and 4.2b show microbenchmark execution time and dynamic energy breakdown, respectively. Figures 4.3a and 4.3b show real application execution time and dynamic energy breakdown, respectively.

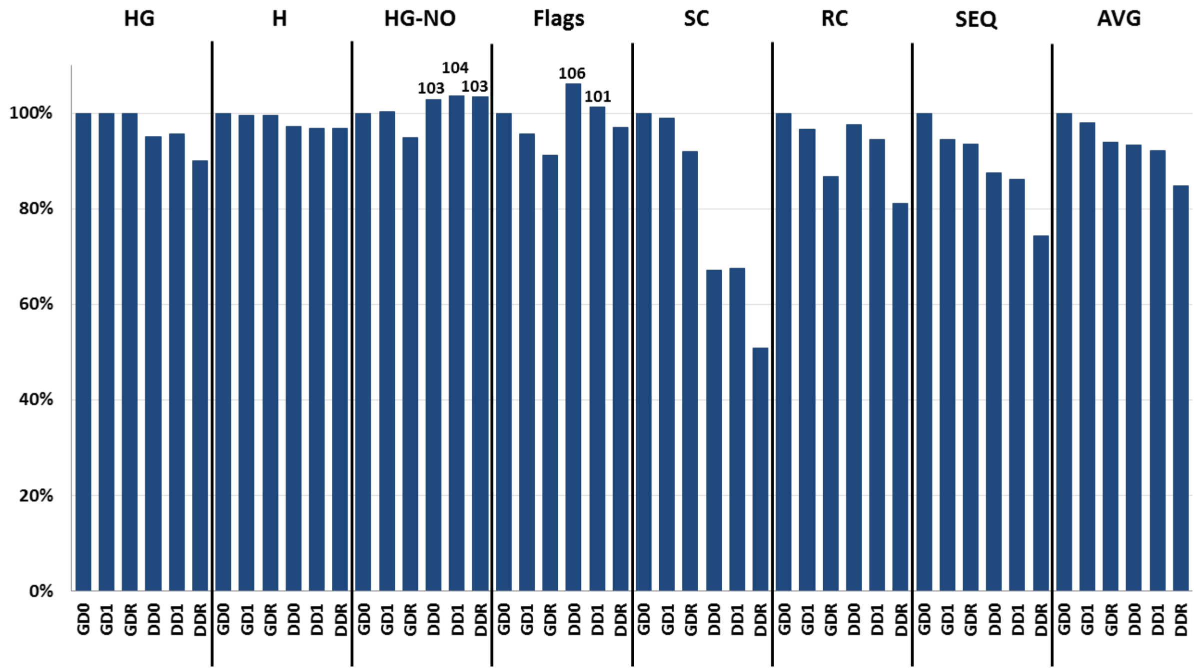
The relaxations enabled by DRF1 benefit the real applications more than the microbenchmarks, since the microbenchmarks are composed primarily of atomic accesses and avoiding synchronization actions only improves cache reuse for data accesses. Across all microbenchmarks and benchmarks, DRF1 reduces execution time for GPU coherence (DeNovo) by on average 11% (11%) and dynamic energy by 10% (12%) relative to DRF0.

With DRFrlx, multiple atomics can be issued concurrently by a thread, increasing memory level parallelism. This reduces execution time for GPU coherence (DeNovo) by on average 9% (7%) by enabling more memory level parallelism for atomics. However, the improved parallelism does not affect cache hit rate or reduce the number of requests sent. Therefore, DRFrlx has a minimal effect on energy for both GPU coherence and DeNovo.

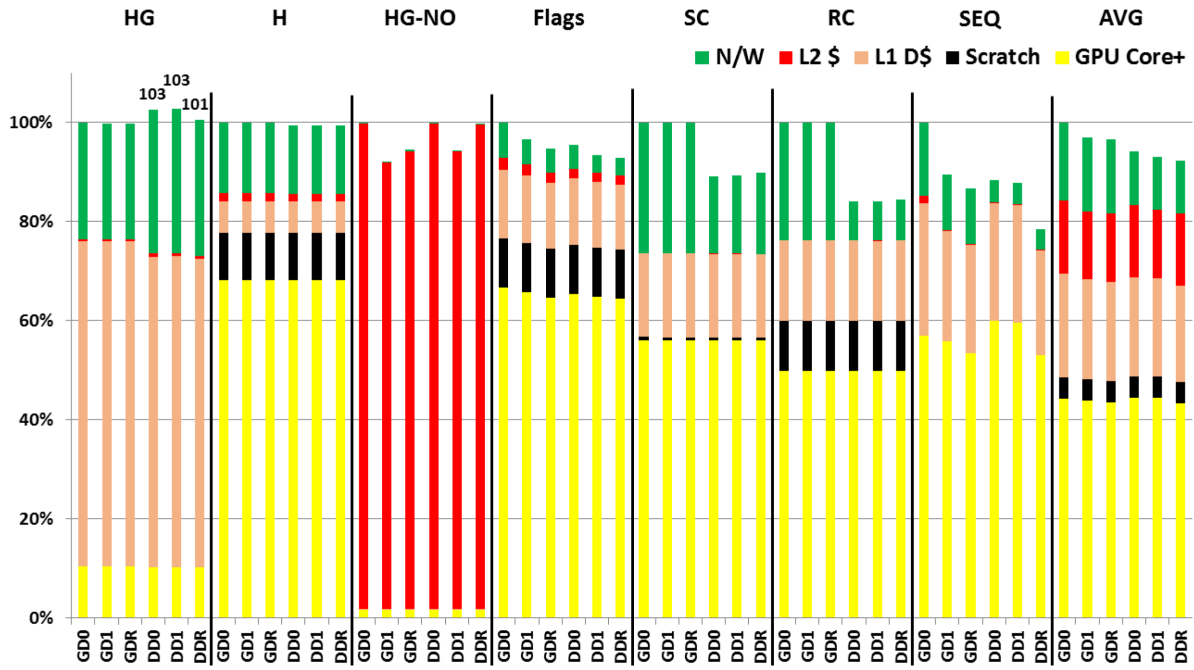
For both microbenchmarks and real applications, DeNovo's improved ability to exploit locality in written data and atomics tends to build upon the benefits of atomic relaxations. Using DeNovo rather than GPU coherence results in an average reduction in execution time (energy) of 14% (16%), 14% (18%), and 11% (18%) for DRF0, DRF1, and DRFrlx respectively.

4.6 Summary

Atomic relaxation has long been used to improve performance in CPU codes, and it is arguably even more important for software-managed coherence strategies like GPU coherence. Although there are many cases where relaxed atomics can be used safely and effectively, existing DRF memory models offer confusing and often poorly defined semantics for relaxed atomics when they are used in a program. With DRFrlx, we identify these safe use cases of atomic relaxation, formalize what makes them safe, and offer SC-centric semantics to programs that use them in this way. With DRFrlx, heterogeneous systems can use relaxed atomics to relax ordering constraints and elide flush and invalidate actions without sacrificing a simple DRF memory model.

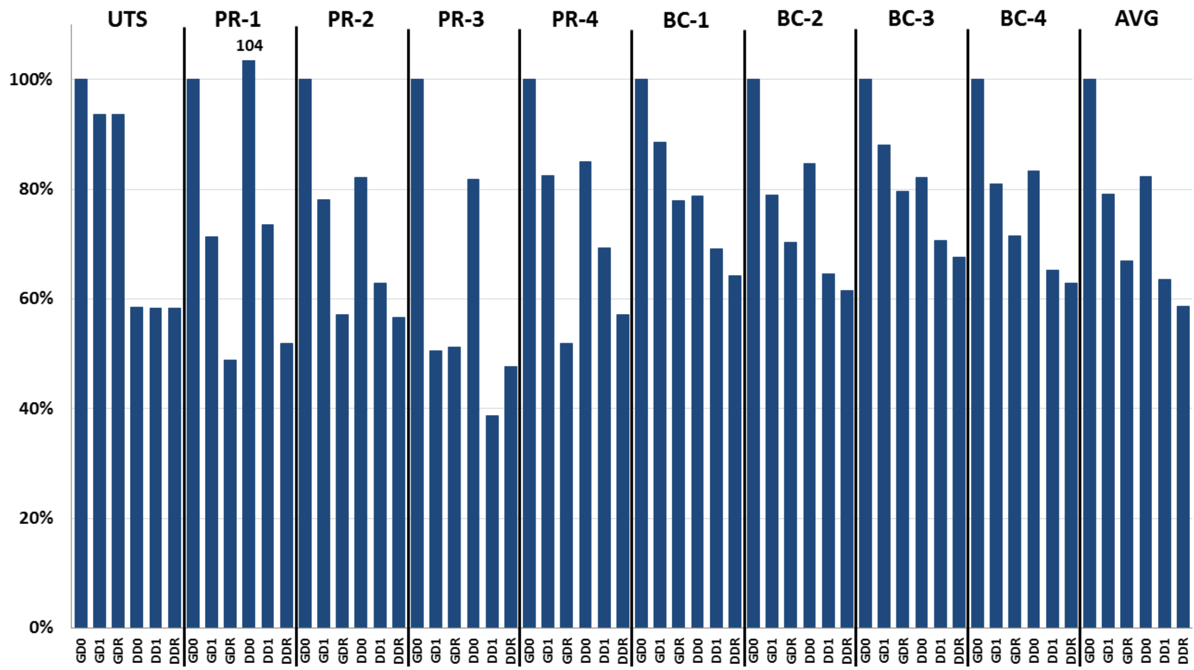


(a) Execution time

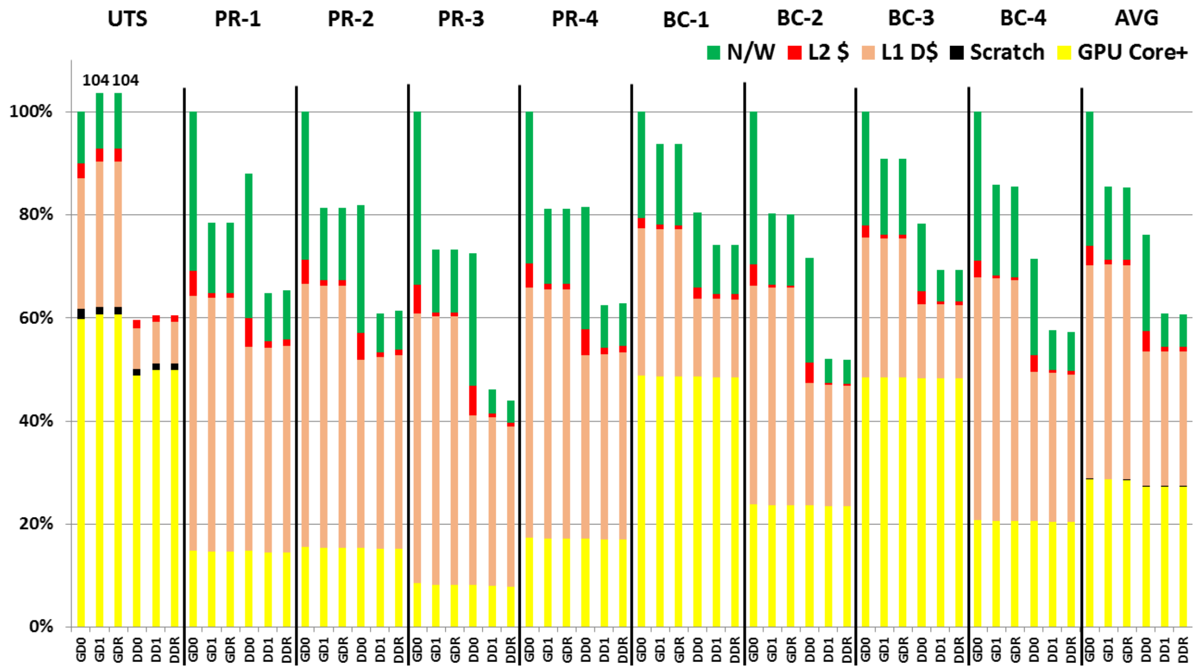


(b) Dynamic energy

Figure 4.2: Execution time and energy for microbenchmarks, normalized to *GD0*.



(a) Execution time



(b) Dynamic energy

Figure 4.3: Execution time and dynamic energy for real applications, normalized to *GD0*.

DRFrlx may not be sufficient for every conceivable relaxation use case. For patterns where relaxation can lead to tolerable non-SC behavior, DRFrlx is particularly restrictive. Only quantum atomics may be used here, and they must be used in a controlled way that avoids introducing illegal races in the quantum-equivalent program. Since quantum loads are effectively replaced with a call to a random function in a quantum-equivalent program, this limits how the value returned by a quantum load may influence control flow or future memory accesses. For example, if the result of a quantum load is used to calculate the address of a subsequent memory access, the subsequent access in a quantum-equivalent program could illegally race with a concurrent access to any potential address that may have been generated from a call to rand(). It can be argued that these types of accesses should not use relaxed atomics because they introduce unacceptable complexity, and DRFrlx is therefore sufficient. Even so, the incremental design of the DRFrlx specification allows for further refinement, and incorporating a broader set of safe and useful relaxed access patterns is one possible future research direction.

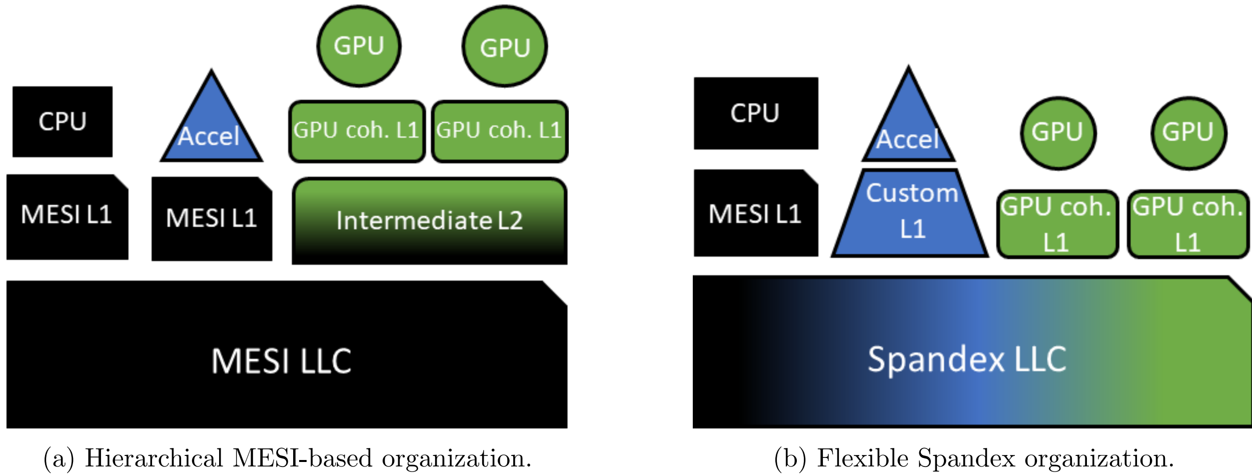
Chapter 5

Spandex: Efficiently Integrating Heterogeneous Coherence Strategies

As we detail in prior sections, when implementing heterogeneous coherence, conventional coherence and consistency strategies that work well for CPUs do not always work well for GPUs, and vice versa. Devices in heterogeneous systems can have a wide range of memory demands which in turn motivate different cache coherence strategies. Efficiently interfacing devices which prefer different strategies into a single coherent system poses a significant challenge as systems become more specialized and exhibit increasingly diverse memory demands. Existing integration solutions tend to connect devices using a MESI-based protocol, sometimes with an intermediate cache level for connecting device types which are incompatible with MESI. Figure 5.1a illustrates this organization. In this chapter we propose Spandex, a coherence interface which is flexible enough to efficiently integrate devices and workloads with widely varying memory demands and coherence strategies. As illustrated in Figure 5.1b, Spandex can directly interface diverse coherence strategies, adapting to suit the memory demands of a wide range of workloads. Although the Spandex design and evaluation focuses on interfacing diverse workloads on a single chip, its simplicity and scalability make it an attractive candidate for multi-chip coherence as well.

5.1 Background

When choosing a coherence strategy for a device, there are three fundamental design decisions to make: 1) how stale data is invalidated, 2) how writes are propagated, and 3) what granularity is used for state tracking and communication. In Section 5.1.1 we begin by classifying three coherence protocols described in prior sections – MESI, GPU coherence, and DeNovo – in terms of these design dimensions (summarized in Table 5.1). Based on this classification, we discuss the memory demands and locality properties that could make a device or workload a good fit for each. In Section 5.1.2



(a) Hierarchical MESI-based organization.

(b) Flexible Spandex organization.

Figure 5.1: Comparison of Spandex and a common existing integration strategy.

Table 5.1: Coherence strategy classification.

Coherence Strategy	Stale invalidation	Write propagation	Granularity
MESI	writer-invalidation	ownership	line
GPU Coherence	self-invalidation	write-through	loads: line stores: word
DeNovo	self-invalidation	ownership	loads: flexible stores: word

we describe some of the most common solutions for integrating these diverse coherence strategies in heterogeneous system.

5.1.1 Protocol Classification

MESI-based protocols (including MOESI, MESIF, etc.) are designed to exploit as much locality as possible by using **writer-initiated invalidation**, **ownership**-based (write-back) caches, and **line granularity** state and communication. Writer-initiated invalidation means that every read miss triggers a request for read permission in the form of Shared state – this permission is revoked only when a future write to the line sends an explicit invalidation to the reader. Ownership-based caching means that every cache miss for a write or atomic read-modify-write (RMW) access triggers a request for exclusive write permission in the form of ownership, or Modified state. Line granularity means that both read and write permissions are requested for the full line in MESI. Once read permission (Shared state) or write permission (Modified or Exclusive state) is obtained,

subsequent reads or writes to the same cache line may hit in the cache until a conflicting access from a remote core causes a downgrade or until the cache line is evicted. This strategy can offer high cache efficiency by exploiting temporal and spatial locality, but it comes at a cost.

The overhead incurred by writer-initiated invalidation imposes throughput and scalability limitations on the system. Each write miss must trigger invalidations in every core that may have read or write permissions for the target cache line. This can incur significant latency, storage, and communication bottlenecks as core counts and memory request throughput demands increase.[1] Tracking state at cache line granularity rather than word granularity helps reduce this overhead. However, it also increases the likelihood of false sharing, which causes wasteful communication, latency, and state downgrades when cores access different words in the same cache line.

In addition, MESI-based protocols also suffer from high complexity. MESI is a read-for-ownership (RfO) protocol, which means that servicing a request for the Modified state requires transferring ownership as well as providing up-to-date data. On a write miss, caches must transition to a transient blocking state, typically delaying subsequent requests to the target cache block while all remote owners or sharers are downgraded and up-to-date data is retrieved. These transient states degrade performance, add complexity, and make extensions and optimizations to MESI-based protocols difficult to implement and verify.

MESI's limited scalability, high complexity, and ability to exploit locality make it a good fit for CPU or accelerator workloads with small core counts, high locality, memory latency sensitivity, and low memory bandwidth demands.

GPU coherence, in contrast, is designed for high throughput and simplicity. Rather than obtaining ownership for writes, GPU L1 caches **write-through** dirty data to the backing cache. Similarly, atomic read-modify-write (RMW) operations bypass the L1 and are performed directly at the backing cache. Rather than sending invalidation messages to potential sharers on a write miss, GPU caches rely on software cues (kernel boundaries or synchronization accesses) to **self-invalidate** potentially stale data in the local cache at appropriate points. Finally, request granularity is chosen to exploit spatial locality while minimizing coherence overheads: read requests are sent at **line granularity** while write-through and RMW requests are sent at the granularity of updates or

[1]Which of these overheads dominates depends on the sharing patterns of the target workload, and whether a snoopy or directory-based protocol is used.

word granularity.

The primary advantage of GPU coherence lies in its simplicity. By using write-through caches and self invalidation, GPU L1 caches avoid the overheads of obtaining read and write permission, including sharer invalidation, indirection, and transient blocking states. Sending write-through and RMW requests at the granularity of updates avoids the latency and communication overheads of RfO caches. However, GPU coherence can still exploit locality by coalescing stores to the same line in the write buffer. Self-invalidation enables read data to be tracked and communicated at line granularity without the risk of wasteful downgrades due to false sharing. As a result, GPU coherence exploits the abundant spatial locality in GPU workloads while sustaining a higher memory request bandwidth than is possible in a MESI-based protocol.

However, the simplicity of GPU coherence has some disadvantages when it comes to synchronization. As described in Section 3.1, GPU coherence must perform expensive cache flushes and invalidations at synchronization points, which can significantly limit cache efficiency if synchronization is frequent. In fact, modern GPU caches are designed to be relatively small, with the expectation that temporal locality is rare and long-term cache reuse is unlikely. Even with optimizations like scoped synchronization [55], relaxed atomics [52], or hLRC [10], GPU coherence is a poor fit for conventional CPU workloads which exhibit high temporal locality and latency sensitivity. Instead, GPU coherence performs best for workloads where global synchronization is infrequent, temporal locality is absent or reuse distance is small, and memory latency is not a performance bottleneck.

The **DeNovo** protocol can be thought of as a sweet spot between the complexity and cache efficiency of MESI at one end and the simplicity and expensive synchronization actions of GPU coherence at the other. Like MESI, DeNovo coherence obtains **ownership** for stores and atomic accesses. However, like GPU coherence, DeNovo **self-invalidates** stale data at synchronization points. Both read and write requests are sent (and Owned state is tracked) at **word granularity**, although a read response may be sent at line granularity when more data in the requested line is available (Owned) at the responding core. By requesting and tracking ownership at modification granularity, DeNovo avoids false sharing. In addition, writes do not need to request up-to-date data and ownership can be transferred without transient blocking states delaying subsequent requests.

Since Owned data is not invalidated at synchronization points, DeNovo caches are able to exploit

reuse in this data even in the presence of frequent synchronization. Thus, DeNovo is able to achieve better cache efficiency than GPU coherence while avoiding much of the coherence overhead, false sharing, and transient states that come with writer-initiated invalidation and RfO protocols like MESI. This makes DeNovo a good fit for a wide range of CPU, GPU, and accelerator workloads.

Of course, DeNovo is not ideal for every workload. By using word granularity writes, DeNovo is less able to exploit spatial locality in written data (although writes to the same line can be coalesced into a single request in the DeNovo write buffer). Additionally, self-invalidation can still harm cache efficiency if there is locality in non-owned data. Even with optimizations like selective region invalidation, DeNovo performs best for programs with high temporal locality in written data and low locality or predictable sharing patterns in read data.

5.1.2 Heterogeneous Coherence Solutions

Integrating coherence strategies as diverse as MESI, GPU coherence, and DeNovo is an important challenge for coherent heterogeneous systems. In this section we describe some key related work that motivated Spandex and influenced the baseline system for our evaluations (Section 7.2 provides a more comprehensive description of related work).

Existing heterogeneous coherence solutions often rely on an assumption of limited inter-device communication demands. This motivates a MESI-based last level protocol, potentially with an intermediate cache level for filtering requests from devices that do not work well with MESI, such as GPU cores.

For example, the IBM Coherent Accelerator Processor Interface (CAPI) enables FPGAs and other accelerators to use a coherent MESI-based cache which interfaces with a snoopy MESI-based last level cache fabric [108]. This design enables efficient high-bandwidth communication between accelerators which share a cache, but the use of a MESI-based protocol means that sharing patterns unsuitable for MESI (e.g., high-throughput streaming workloads with limited temporal locality) may incur excessive coherence overheads.

The AMD APU uses a hierarchical cache structure with a MESI-based directory to integrate its CPUs and GPUs [21]. This approach is best suited for hierarchical sharing patterns; intra-device communication is prioritized while communication between CPU and GPU cores suffers added latency and energy overhead due to hierarchical indirection and blocking transient states at the

LLC. In our evaluation, the baseline hierarchical MESI configuration is based on this design.

The ARM AMBA Coherent Hub Interface (CHI, [16]) specifies an interface for implementing coherence between CPUs and accelerators. Unlike CAPI, CHI defines "non-snoopable" address regions and request types, which could potentially be used in a similar way as self-invalidated loads and write-through stores. CHI also supports direct integration of CPUs and accelerators, partial cache line validity, "far" (i.e. remotely performed) atomic requests, cache stashing (i.e. producer-consumer forwarding, discussed in more detail in Section 7.4), and the ability to configure devices as home nodes, which adds further flexibility. However, data in a CHI system must be either snoopable or non-snoopable, limiting interactions between devices which may prefer to access the same data with different request types. In addition, although a detailed implementation of a last level directory protocol is outside the scope of the CHI specification, the specified transaction messages required for different request types suggest a blocking MESI-based protocol, which can result in limited throughput scalability, high protocol complexity, and costly transient states for inter-device communication.

Until recently, the assumption of regular or limited inter-device sharing has been valid. Inter-device communication has historically been expensive, so conventional heterogeneous algorithms tend to prioritize intra-device communication when possible. Thus, the inter-device throughput limitations of a MESI-based LLC are unlikely to degrade performance, especially when high-throughput devices interface through an intermediate cache level which can coalesce and filter requests from threads within the device. However, as accelerators become more tightly coupled and heterogeneous applications become more capable and diverse, it is unclear that hierarchical or MESI-based approaches are flexible enough to generalize efficiently to a broader range of access patterns.

5.2 Design

The Spandex design can be divided into device side logic and integration logic. At both the device side and the integration side, Spandex defines a set of supported states, a request interface, and the request handling and state transition logic required to implement Spandex.

The device request interface, described in Section 5.2.1, details the supported device states and the request types used to interface a device. At the integration side, Section 5.2.2 discusses the

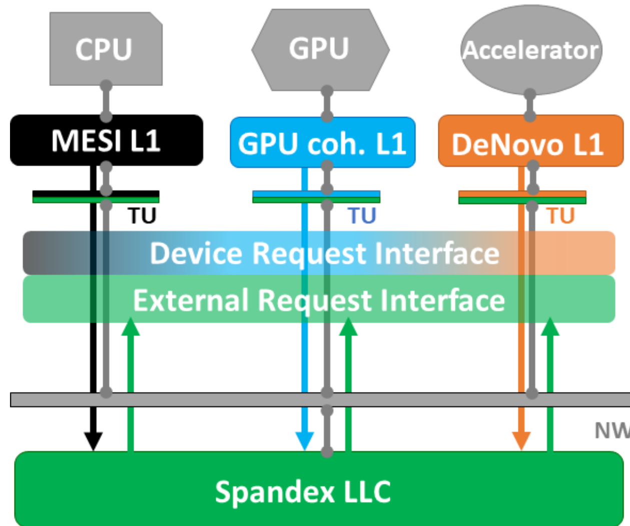


Figure 5.2: Overview of Spandex design. Requests to and from the LLC travel through a generalized network, which may be unordered.

states, requests, and state transitions at the Spandex LLC. Section 5.2.3 describes the external request interface. This is the logic required at the device side to handle forwarded requests and probes from the Spandex LLC, and Section 5.2.4 provides examples of how a thin per-device translation unit (TU) may be used to implement elements of this logic when it is not natively supported by device caches. Section 5.2.5 discusses the memory consistency assumptions of Spandex, and Section 5.2.6 discusses the overheads of Spandex relative to a line granularity MESI-based LLC. Figure 5.2 illustrates how all of the above components fit together.

Throughout the following sections, Figures 5.3a-5.3d are used to illustrate how Spandex handles some basic request types of varying granularity. In each figure a CPU, GPU, and custom accelerator interface directly with the Spandex LLC. Each device has a local cache tailored to the memory demands of that device that connects to the Spandex system through a custom translation unit.

5.2.1 Device Request Interface

Spandex supports four stable coherence states at an attached device memory: Invalid (I), Valid (V), Owned (O), and Shared (S). Although attached devices may use other protocol state names, any internal state should map to one of these supported states from the perspective of the rest of the system. Devices may track state at any granularity. However, requests for state and data may only occur at word or line granularity, and devices must be able to handle responses, forwarded

requests, and probes at word granularity (discussed more in Section 5.2.3). The device states and when it generates a Spandex request are:

- **I** indicates that the data is invalid. A read or write from the device must generate a Spandex request.

- **V** indicates that the data either is up-to-date, or is stale but will be invalidated before a subsequent read. A read hits on Valid data without further action but a write generates a Spandex request. A key attribute of **V** is that the device itself is responsible for self-invalidating valid data at appropriate points to ensure no stale data is read. The consistency model defines when data becomes stale (Section 5.2.5).

- **S** indicates that the data is up-to-date. A read hits without further action but a write generates a Spandex request. **S** is similar to **V** except that a device is not required to self-invalidate data in Shared state. The system (Spandex LLC) is responsible for sending an invalidation message when Shared data becomes stale.

- **O** indicates the data is an exclusive up-to-date copy. A read, write, or atomic RMW hits on this state without further action.

Spandex's flexibility arises from its ability to interface devices with widely varying memory demands. There are 7 request types that may be issued from a Spandex device. Requests carry granularity information (word or line),[2] and multiple word granularity requests to the same line may be coalesced into a single multi-word request with a bitmask indicating the targeted words within the cache line.

The following requests are sufficient to support devices that use write-through or ownership for updates, self-invalidating or writer-invalidated reads, and diverse request granularities. Table 5.2 provides a mapping from GPU coherence, DeNovo, and MESI requests to Spandex request types and granularity. Every Spandex request (Req) type has an associated response (Rsp) type.

- **ReqV** is generated for a self-invalidated read miss; it simply requests the up-to-date data at the target address. A RspV response causes a transition to V state.

[2] Although other request granularities may be useful for some workloads, word and line granularity were sufficient for the devices we considered.

Table 5.2: Type and granularity of requests generated for read misses, write misses, and replacements of owned data in GPU coherence, DeNovo, and MESI caches. *A DeNovo ReqV request is issued at word granularity, but the responding device may include any available up-to-date data in the line.

Device Type	Device Request	Spandex Request	Granularity
GPU Coherence	Read	ReqV	line
	Write	ReqWT	word
	RMW	ReqWT+data	word
DeNovo	Read	ReqV	flexible*
	Write	ReqO	word
	RMW	ReqO+data	word
	Owned Repl	ReqWB	word
MESI	Read	ReqS	line
	Write	ReqO+data	line
	RMW	ReqO+data	line
	Owned Repl	ReqWB	line

- **ReqS** is generated for a writer-invalidated read miss; it requests both up-to-date data and Shared state. A RspS response causes a transition to S state.

- **ReqWT** is generated for a write-through store miss. The granularity of this request type is the same as the granularity of modification, so up-to-date data is not needed to satisfy this request. If target data was previously in I state, a ReqWT operation causes a transition to V state at the time of update.

- **ReqO** is generated for an ownership-based store miss that is overwriting all requested data. Thus, it requests ownership but not the up-to-date data. If target data was in I, V, or S state, a ReqO operation causes a transition to O state at the time of update.

- **ReqWT+data** sends an update operation to be performed at the LLC. This is similar to a ReqWT, but the operation also requires up-to-date data. Unlike ReqWT, which simply overwrites the current data value, this request must specify the required update operation and may be used for an atomic read or RMW performed at the LLC.[3] A RspWT+data response carries the value of the data before the update was performed and triggers a downgrade at the requestor cache (since the response data is potentially stale).

- **ReqO+data** is generated from an ownership-based (write-back) cache for a request that

[3]For GPU coherence, all atomic accesses are performed at the LLC.

needs both the up-to-date data *and* ownership. This may be used to request ownership for a locally performed RMW operation, or for a store in a line granularity ownership-based cache that does not overwrite all data in the line. A RspO+data response causes a transition to O state.

- **ReqWB** writes back owned data to the LLC. This request is necessary whenever Owned data is downgraded as a result of local cache actions (e.g., a cache replacement). During a pending ReqWB, up-to-date data must be retained until the write-back has completed.

Since Spandex tracks ownership at word granularity (see Section 5.2.2), different words within a single multi-word request may be satisfied at different devices. Therefore, a device that can issue multi-word requests (including line granularity requests) must be able to handle multiple partial word granularity responses. The implications of this requirement for line-based protocols are discussed in Section 5.2.4.

5.2.2 Last Level Cache (LLC)

LLC States

There are four stable coherence states at the Spandex LLC: Invalid (I), Valid (V), Owned (O), and Shared (S). I, V, and O come directly from DeNovo while S is needed to support devices that use writer-initiated invalidation.

- **I** indicates that only the backing memory is guaranteed to have an up-to-date copy of the data.

- **V** indicates the data at the LLC is up-to-date and is not in Shared or Owned state in any attached device memory.

- **S** indicates the data at the LLC is up-to-date, but one or more sharer devices may require invalidations if the data is modified or replaced.

- **O** indicates the target data is Owned within an attached device memory.

Spandex tracks ownership at word granularity, and this helps avoid many of the inefficiencies present in MESI-based protocols. In protocols with line granularity ownership state (e.g., MESI), an ownership request must revoke ownership from the previous owner and wait until ownership and data for the full line have been transferred. This is especially wasteful in the case of false sharing, when the devices are accessing different words in the same line. With word granularity ownership state, devices can issue ReqO and ReqWT requests for exactly the words being updated in a line. These requests can be satisfied without obtaining up-to-date data (because the requested data is overwritten), or revoking ownership for the entire block.

Figure 5.3a demonstrates these benefits. The accelerator device issues a word granularity ownership request (①) which triggers an immediate transition to owned state and a data-less RspO for the target words (②). The GPU then issues a write-through request for disparate words in the same line (③), resulting in an immediate update of the LLC data and a data-less RspWT to the requestor (④). Due to the word granularity ownership tracking, false sharing is avoided and write-only requests proceed without requiring blocking states at the LLC or data responses.

To limit tag and state overhead, allocation occurs at line granularity. For each line, two bits indicate whether the line is Invalid, Valid or Shared. For each word within the cache line, a single bit tracks whether the word is Owned in a remote cache. For each Owned word, the data field itself stores the ID of the remote owner (this is similar to how DeNovo tracks owners).

Implementing byte granularity ownership state at the LLC is also possible, but it would incur more overhead (a bit per byte vs. a bit per word). Based on the workloads we studied, byte granularity stores that cannot be coalesced with others into a full word store are expected to be rare, so the benefits of byte granularity state tracking do not appear to be worth the overheads. Spandex therefore requires byte granularity stores to use word granularity ReqWT+data or ReqO+data rather than ReqWT or ReqO requests to ensure non-modified data in the requested word remains up-to-date.

LLC Requests

In addition to coherence requests generated at the attached devices, the LLC itself may initiate requests when necessary for downgrading the state in remote owner or sharer devices.

- **RvkO** is used to revoke ownership from an owner device and trigger a write-back of the

Table 5.3: The state transition triggered at the LLC by each request type (Next State) and the request type forwarded to the owning core in the event the data is in O state (Fwd Msg). An entry of – indicates no transition or forwarded request is necessary.

Request Type	Next State	Fwd Msg
ReqV	–	ReqV
ReqS (1)	S	ReqS
ReqS (2)	–	ReqV
ReqS (3)	O	ReqO+data
ReqWT	V	ReqO
ReqO	O	ReqO
ReqWT+data	V	RvkO
ReqO+data	O	ReqO+data
ReqWB from owner	V	–
ReqWB from non-owner	–	–

owned data, and has a corresponding response type of **RspRvkO**.

- **Inv** is used to invalidate shared data in a sharer device and has a corresponding response type of **Ack**.

LLC State Transitions

The Spandex LLC serves as the coherence point for all device caches and serializes all write requests (ReqWT[+data] and ReqO[+data]) to a given (coherent) data address. Table 5.3 describes, for each device request type, the next stable state at the LLC and, in case the initial state is O, the message the LLC must forward to the owner. ReqS requests can be handled in multiple ways (see discussion of Shared state below). For multi-word requests, each word is handled individually, potentially triggering different actions.

In the common case, request handling occurs immediately without any blocking states. For all requests, if the target data is in V state at the LLC, the LLC immediately satisfies the request, triggers a state transition, and responds to the requestor. If a read request (ReqV or ReqS) arrives for target data in S state, the LLC immediately responds to the requestor and, in the case of ReqS, updates the sharer list. If the target data is in O state in the LLC, the LLC immediately forwards the request to the owning core rather than responding to the requestor. All requests other than ReqWT+data and ReqS (1) for data in O state trigger an immediate state transition.

For some state transitions, blocking states are needed to wait for downgrades and write-backs to

complete. Specifically, if a write request (ReqWT[+data], ReqO[+data], ReqWB) arrives for target data in S state, the LLC must send Inv requests to any potential sharer devices and transition to a blocking state while waiting for Ack responses. If a ReqS (1) or ReqWT+data request arrives for target data in O state, the LLC transitions to a blocking state while it waits for the forwarded request to trigger a write-back from the owning core. In both cases, once all Acks have been collected or the write-back has completed, the data transitions to the next stable state specified in Table 5.3.

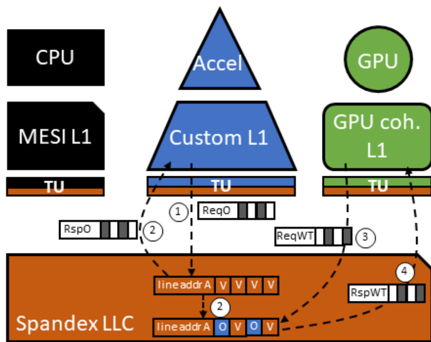
Figure 5.3b illustrates how a word granularity ReqWT+data request (①) is handled when the target data is in O state at the LLC. Since a ReqWT+data requires both up-to-date data and write serialization, the LLC sends a RvkO request to the owning accelerator device and transitions to a transient state *tr* (②). After receiving the RvkO, the owner device responds with a RspRvkO request for the entire line (③). The LLC performs the requested update and responds to the GPU and MESI caches with a RspWT+data and RspWB, respectively (④).

Supporting Shared State

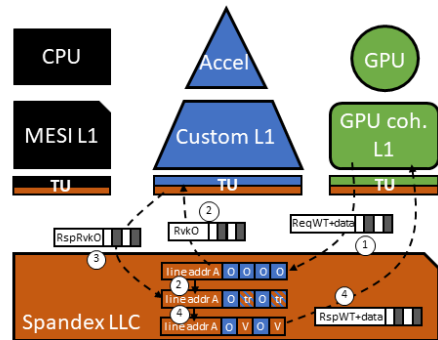
To efficiently integrate devices that use writer-invalidated reads (e.g., MESI caches), Spandex extends the DeNovo protocol with S state. However, a writer-invalidated read (ReqS) does not always need to trigger a transition to S state. Spandex can handle a ReqS request in multiple ways, as Table 5.3 illustrates, depending on the system's demands and constraints.

The LLC may implement writer-initiated invalidation and transition to S state, represented by option (1). Supporting writer-initiated invalidation can improve reuse for data that is concurrently read by multiple writer-invalidated device caches. However, it also incurs some complexity and overhead in the protocol. A ReqS (1) for data in O state triggers a transition to a blocking state while the current owner completes a writeback. A write request (ReqO[+data], ReqWT[+data]) for data in S state triggers a transition to a blocking state while potential sharers are invalidated. In addition, tracking and invalidating sharers incurs storage and network traffic overheads. These overheads are consistent with the overheads of writer-initiated invalidation in MESI.

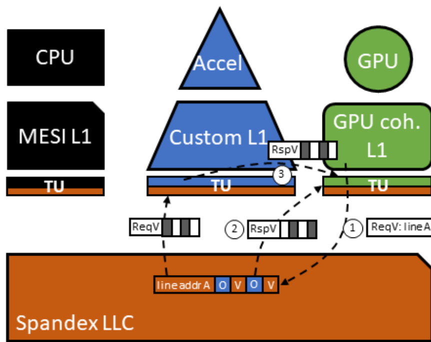
Alternatively, a ReqS may be treated the same as a ReqV or ReqO+data request, represented by options (2) and (3) in Table 5.3, respectively. Both options avoid the complexity and overheads of Shared state. However, option (2) requires the requesting cache to downgrade the target data



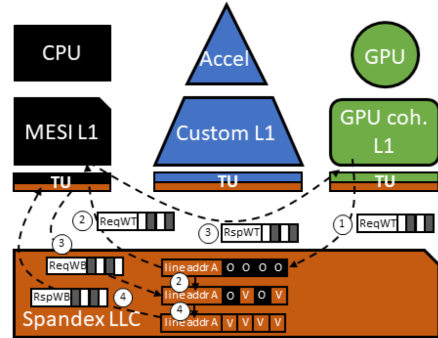
(a) Handling word granularity ReqO and ReqWT



(b) Handling word granularity ReqWT+data for remotely owned data



(c) Handling line granularity ReqV



(d) Handling word granularity ReqWT with line granularity owner

Figure 5.3: Handling basic request types at the Spandex LLC.

Table 5.4: The state transition and response message triggered at a device by each external Spandex request. If the target data is not in the expected state when a request arrives, different behavior may be required (discussed in Section 5.2.3).

Spandex Request	Expected State	Next State	Response
ReqV	O	O	RspV to requestor
ReqO	O	I	RspO to requestor
ReqO+data	O	I	RspO+data to requestor
RvkO	O	I	RspRvkO to LLC
Inv	S	I	Ack to LLC
ReqS	O	S	RspS to requestor RspRvkO to LLC

to Invalid after the read operation is satisfied and therefore precludes any further reuse in the requestor cache. In contrast, option (3) requires the requesting cache to upgrade the target data to Owned state after the read operation is satisfied. This enables additional reuse in the requestor cache, but precludes multiple concurrent readers and can lead to wasteful ownership transfers if there is high contention for the target data.

In our evaluation, Spandex uses option (1) if the target data is in S state or owned in a MESI core. In all other situations (I, V, or owned at a non-MESI core), we use option (3). This is similar to MESI's response to a Shared request with Exclusive state if data is not present in another MESI cache.

5.2.3 External Request Interface

We next describe how a Spandex device must handle external messages (forwarded requests and probes) it may receive from the Spandex system. Table 5.4 summarizes, for each external request type, the expected device state of the target data, the next stable state, and the response sent for each external request type. A device must be able to handle an external request type if it supports the expected state for that request. Since Spandex tracks ownership at word granularity (see section 5.2.2) and external requests may therefore arrive at word granularity, devices must be able to implement these transitions and responses at word granularity. Section 5.2.4 discusses how the per-device TU can be used to implement this requirement in line granularity caches.

It is possible that the device state will not match the expected state of the specified transition at the time the request arrives. This happens when data requests race with other requests or write-

backs to the same data, and different reasons can require different handling. When an external request arrives, the target data may be (1) in a pending transition to the expected state, (2) in a pending transition away from the expected state, or (3) specifically for an Inv request or a forwarded ReqV request, in a stable state other than the expected state. The behavior required for cases (1) and (2) is consistent with how conventional protocols handle race conditions. Case (3), however, involves a race that is unique to Spandex and thus requires additional consideration.

1) Pending Transition to Expected State: If an external request requires a data response (ReqV, ReqS, ReqWT+data, ReqO+data, or RvkO request) and up-to-date data is not available but is pending (possible for a pending ReqO+data), the external data request must be delayed until the device's pending data request completes. In many cases, however, up-to-date data is either available (the pending request is a ReqO) or unneeded (the external request is a ReqO or Inv), and a response to the external request may be sent immediately.

2) Pending Transition from Expected State: The required device behavior depends on whether its pending transition is an upgrade or a downgrade. If it is an upgrade, then it must be a pending transition from S to O (since external requests expect data in S or O state, and no upgrade is possible from O) with an external Inv request. In this case no state transition is necessary and the external request will not require up-to-date data; the device should respond immediately, and the pending upgrade may proceed as normal.

If the pending transition is a downgrade from the expected state, it must be a transition from O to I due to a pending ReqWB (all other downgrades from O or S happen immediately). Here the device may respond immediately, with data if necessary, to the forwarded request.[4] If the external request itself triggers a downgrade (ReqO[+data], RvkO, ReqS), it should also trigger completion of the pending ReqWB since the LLC no longer considers this device the owner.

3) Stable State other than Expected (Inv or ReqV): Inv requests differ from other request types because they expect a state that can be silently downgraded on the device side. If a device receives an Inv request for data in a stable state other than S, it may simply Ack the request without updating its state.

ReqV requests differ from other request types in that they do not affect coherence state at the

[4]In the case of a RvkO or ReqS, the RspRvkO does not need to carry data since up-to-date data has already been sent in the pending ReqWB.

LLC or the owning core and they enforce no global ordering with other operations to the same target data. As a result, when a ReqV request is forwarded to an owning core, that core may completely transition away from Owned state before the forwarded ReqV arrives since neither the owning core nor the LLC has any way to know there is a ReqV request en route for the owned data. When this happens, Spandex requires that the incorrectly assumed owner Nack the failed ReqV request, and the requesting device retry the ReqV. This is the same strategy used by DeNovo.

However, Spandex faces a new challenge not encountered by MESI, GPU coherence, or DeNovo alone: ReqV starvation. Only Spandex allows line granularity ownership requests from MESI caches to race with ReqV requests from GPU coherence and DeNovo caches. When a ReqV request races with frequent ownership requests, a rapidly changing ownership state can lead to repeated failure and eventual starvation for the ReqV access.

To avoid ReqV starvation, after a finite number of failed ReqV requests (1 in our evaluation), a Spandex device must replace the failed ReqV request with a request type that will enforce ordering for racy accesses: a ReqWT+data or ReqO+data request. Both request types enforce a global ordering of operations to a given address in the presence of racy ownership requests and ensure forward progress.

5.2.4 Translation Unit Responsibilities

Although much of the required functionality described in section 5.2.3 is implemented natively by attached device caches, the per-device translation unit (TU) is responsible for filling in any gaps. A TU is customized to conform to external device IP, which may have its own cohererence interface and request types, to the Spandex interface. It may be incorporated into the device IP itself by the device vendor, it may be a separate structure designed by the device vendor, or it may be designed by a third party based on detailed information of the device interface. Generally, tighter integration with the target device will enable better efficiency since the TU logic would have access to more relevant internal device state (e.g., the MSHR). In our evaluation, a separate structure is used which sits between the device cache and the network. It has the ability to intercept, delay, or insert requests on these connections, and in some cases the TU must insert basic queries for the current value or state of data in the device cache. Here we describe how a discrete TU helps to implement the required functionality for GPU coherence, DeNovo, and MESI caches in a Spandex

system.

The states used in GPU coherence (I, V) and DeNovo (I, V, O) map directly to Spandex states I, V, and O. MESI states I and S similarly map directly to Spandex states I and S, while M and E both map to O state.

Required functionality for GPU coherence TU: Since GPU coherence does not support O or S state, its caches do not need to handle forwarded requests or probes. However, GPU coherence alone does not support ReqV retries, so the TU must retry a Nack for a ReqV and retry the request as a ReqWT+data. In addition, GPU coherence can issue multi-word requests but may not handle partial word granularity responses. Thus, the TU must collect and coalesce responses from multiple sources before sending a response to the GPU cache.

In Figure 5.3c, the GPU coherence TU must coalesce word granularity responses for a line granularity request. First, the GPU sends a line granularity request for valid data to the LLC (①). The LLC immediately responds with valid data in the line and forwards a word granularity request for remotely owned words (②). No state transition is necessary. The owner responds directly to the requestor with valid data (③). The GPU's TU coalesces the responses and sends the GPU cache a line granularity response once all have arrived.

Required functionality for DeNovo TU: The word-based DeNovo protocol already handles partial responses for multi-word requests, as well as responding to forwarded requests for owned data at word granularity. The only added functionality needed at the TU is the ability to replace a Nacked ReqV request with a ReqWT+data or ReqO+data request after a finite number of tries (DeNovo alone does not need to do this).

Required functionality for MESI TU: MESI requires the most help from the TU of the three protocols discussed. Like GPU coherence, the TU must collect and coalesce responses from multiple sources before sending a response to the line-based MESI cache. MESI devices can natively handle ReqS and Inv requests. Since a MESI cache supports O state, it must also be able to handle external requests for owned data at word granularity, which is complicated by MESI's line granularity state. Depending on the state of the target data when the external request arrives, there are three cases discussed in detail below: (1) the data is in stable O state, (2) there is a pending request to bring the data in O state, and (3) there is a pending write-back request for the target data.

1) O state: Requests that require word granularity data or ownership downgrades are converted to line granularity and passed through to the MESI cache. If the request required ownership downgrade for only part of the line, the TU must trigger a ReqWB for any non-downgraded words. Figure 5.3d illustrates this case. First, the GPU sends a word granularity write-through request for data that is remotely owned at the MESI cache (①). The LLC immediately updates the ownership state and data of the written words (to V) and forwards the write-through request to the MESI owner (②). The MESI cache downgrades the requested data, responds directly to the requestor, and triggers a write-back for the words that were not requested in the downgraded line (③). Lastly, the LLC handles the ReqWB and responds (④).

2) Pending O request: The TU is responsible for delaying word granularity requests that require data (ReqV, ReqS, ReqO+data, RvkO) and responding immediately for requests that only require ownership downgrade (ReqO). After ownership for the full line has been received, if any downgrade requests have been received, the TU must cause a transition to I state rather than O state for the target line in the MESI cache, and trigger a ReqWB for any words in those lines that had not received downgrade requests.

3) Pending write-back: The TU is responsible for responding to data and ownership downgrade requests for the data being written back. Ownership downgrade requests are handled as write-back responses by the TU.

As previously discussed a forwarded ReqV may arrive while the MESI device is not in any of the above states. In that case, the TU is required to Nack the request.

5.2.5 Consistency Requirements

The memory consistency model implementation requirements can be divided into those that are met within a device (e.g., preserving the program order of certain memory operations) and those that are met at the system level (e.g., ensuring a write appears atomic such that its value is not visible to a core while other cores may still see the stale value). The exact actions depend on the consistency model used and addressing them for general models is outside the scope of this work. Lustig et al. [77] offer a framework for determining exactly what ordering constraints need to be added when interfacing devices with different consistency models. Here we assume a DRF model.

Since synchronization accesses indicate possible inter-device communication in a DRF program (through happens-before), DRF may be implemented in a Spandex system by 1) restricting the reordering (or concurrent issue) of synchronization accesses with other accesses in program order, 2) completing write buffer (ownership or write-through) flushes at synchronization points, and 3) self-invalidating valid data at synchronization points. Techniques such as DeNovo regions [42], scoped synchronization [55], relaxed atomics [101, 52], and hLRC [10] may relax these consistency requirements for some synchronization accesses. A Spandex system where the devices obey the above requirements and the LLC ensures serialized and atomic writes can be shown to be DRF compliant.

5.2.6 Overheads

The added request type flexibility offered by Spandex may increase the number of message type identifier bits by at most 1 relative to a MESI-based protocol. Word granularity ownership also incurs overheads. An additional state bit per word is needed in the LLC to indicate owned words, a bitmask is needed for multi-word requests to indicate target words in the line (although this overhead may be offset by sub-line data transfer), and multiple forwarded requests and responses are needed when words in a line are owned at different locations (although this overhead may be offset by reduced false sharing). Requests that need to look up the remote owner of some data in the LLC can also be more expensive, since this involves an access to the data block rather than just cache state. While this is expected to increase energy per access, the controller occupancy incurred by such a request is not fundamentally different from an analagous request in a more conventional directory structure, which must read and/or write cache state. We therefore expect comparable bandwidth and request latency to be achievable with sufficient banking and pipelining in the Spandex LLC. Spandex also requires inclusivity for Owned data at the Spandex LLC, and a data block must therefore be allocated whenever data is remotely owned. It is possible to implement a state-only Spandex LLC (avoiding the need for data block allocation), but it would still need to track owner ID at word granularity. As a result, a state-only Spandex LLC cannot achieve the same storage efficiency as a state-only directory for a line granularity protocol.

Many states used in MESI-based protocols such as MESI Exclusive (E), MOESI Owned (MO), or MESIF Forward (F) state are not directly supported in Spandex; these must map to Spandex's O

or S state. It is possible to add support for these states in Spandex, but with some added complexity that is mostly well understood in coherence protocol design. Our goal here is instead to show we can integrate flexibility in diverse protocol dimensions that are emerging as more fundamentally important for emerging workloads in heterogeneous systems.

The overhead incurred by the TU will be highly dependent on the functionality supported by the attached device, and it may require slight changes to device cache IP to enable probes or state updates from the TU. We do not provide area or power cost estimates for the TU, but we expect its cost to be comparable to an MSHR. We do, however, model TU queuing latency in our evaluation, assuming a single-cycle lookup.

Overall, these overheads are expected to be offset by the advantages of Spandex. By supporting significant request flexibility and tracking ownership at word granularity, Spandex avoids the need for a hierarchical cache structure and offers reduced complexity, blocking states, and false sharing in the protocol. This leads to fewer state transitions, less network traffic, and lower latency for a wide range of sharing patterns.

5.3 Methodology

To evaluate Spandex we run a set of CPU-GPU applications on the same integrated architectural simulator used in Sections 2.2, 3.2.3, and 4.5, although using different coherence protocols and 16 CPU and GPU cores (each mesh network node has one GPU core and one CPU attached to it). We compare Spandex against a hierarchical cache structure, each with a variety of CPU and GPU coherence strategies.

5.3.1 Cache Configurations

Table 5.5 describes the 6 memory configurations we evaluate classified by LLC protocol, CPU cache protocol, and GPU cache protocol. Configurations with hierarchical MESI LLC (H-MESI) use a hierarchical cache structure in which GPU L1 caches interface with each other through a shared intermediate L2 cache, and CPU L1 caches interface with the GPU L2 cache through a shared MESI LLC. Configurations with a Spandex LLC directly interface CPU L1 caches and GPU L1 caches through the shared Spandex L2. System parameters for these configurations are given in

Table 5.5: Simulated cache configurations.

Cache Config.	LLC Protocol	CPU L1 Protocol	GPU L1 Protocol
HMG	H-MESI	MESI	GPU coherence
HMD	H-MESI	MESI	DeNovo
SMG	Spandex	MESI	GPU coherence
SMD	Spandex	MESI	DeNovo
SDG	Spandex	DeNovo	GPU coherence
SDD	Spandex	DeNovo	DeNovo

Table 5.6. We do not evaluate a hierarchical Spandex configuration. Although such an organization is possible and may even be preferable for some workloads, our focus is on Spandex's ability to efficiently interface devices in configurations not possible for existing approaches.

We vary the CPU and GPU L1 cache protocols based on what is practical for each device type and what is supported by the cache hierarchy. CPU L1 caches use MESI or DeNovo, while GPU L1 caches use DeNovo or GPU coherence. The hierarchical MESI LLC only supports MESI CPU caches, but its intermediate GPU L2 can support either DeNovo or GPU coherence requests from GPU L1s. Spandex supports MESI, DeNovo, or GPU coherence requests at the LLC. For caches that use self-invalidation protocols, V data is invalidated in a single-cycle flash operation. In SDG, CPU caches depart slightly from the DeNovo protocol, performing atomic accesses at the L2 rather than obtaining ownership (ReqWT+data rather than ReqO+data requests). By matching the CPU atomic access strategy to the GPU strategy, we avoid blocking states that would otherwise arise from inter-device synchronization.

5.3.2 Benchmarks

Synthetic Microbenchmarks

Our synthetic microbenchmarks are a set of simple multithreaded CPU kernels and GPU kernels that share data between devices in a way that highlights the performance implications of using a hierarchical vs. flat cache structure, ownership vs. write-through requests for updates, and writer-invalidated reads vs. self-invalidated reads. While existing applications are generally designed with a fixed coherence strategy in mind, these microbenchmarks highlight a range of sharing patterns that may benefit if systems are given more flexibility in how coherence is implemented.

Table 5.6: Simulated heterogeneous system parameters.

CPU Parameters		
Frequency	2 GHz	
Cores	8	
GPU Parameters		
Frequency	700 MHz	
CUs	16	
Memory Hierarchy Parameters		
Parameter	**Hierarchical**	**Spandex**
L1 Size (8 banks, 8-way assoc.)	32 KB	32 KB
L2 Size (16 banks, NUCA)	4 MB	8MB
L3 Size (16 banks, NUCA)	8 MB	–
Store Buffer Size	128 entries	
L1 MSHRs	128 entries	
L1 hit latency	1	
Remote L1 hit latency in cycles	35−83	
L2 hit latency in cycles	29−61	
L3 hit latency in cycles	CPU: 29−61 GPU: 52−100	–
Memory latency in cycles	CPU: 197−261 GPU: 222−306	197−261

Indirection: In this microbenchmark, the CPU and the GPU take turns transposing a matrix in a loop. CPU threads read tiles in matrix A and write tiles in matrix B, while GPU threads read tiles in matrix B and write tiles in matrix A. Accesses are strided to reduce spatial locality, and tile size is selected to ensure data is not reused from the L1 cache. Indirection primarily demonstrates the cost of hierarchical indirection.

ReuseO: In this microbenchmark, CPU threads sparsely read matrix A and densely read and write matrix B, and GPU thread blocks sparsely read matrix B and densely read and write matrix A. Tiles are sized to fit in the cache, and the process is repeated iteratively so that data written in one iteration is reused at the same core in the subsequent iteration. ReuseO highlights the benefits of using ownership for updates.

ReuseS: In this microbenchmark, CPU threads and GPU thread blocks take turns densely reading and sparsely writing a shared matrix. It highlights the benefits of writer-initiated invalidation because only Shared state can exploit reuse in read data across iterations; data in Valid state must be invalidated in case it has been updated by a remote compute unit.

Applications

We also select a diverse set of applications from Pannotia [38] and Chai [53]. These emerging applications use shared memory to collaboratively execute real-world functions in a CPU-GPU system. From Pannotia we use two iterative graph analytics algorithms: Betweenness Centrality (BC) and PageRank (PR). BC is a push-based algorithm that computes the centrality of each vertex in a graph based on the shortest path to every other node. Each thread updates the neighbors of its assigned nodes and the updates must use atomics since multiple threads may attempt to update the same neighbor. PageRank is a pull-based algorithm that iteratively computes a ranking for every node in a graph based on its neighbors rankings. Threads only update their assigned nodes and only read the values of neighboring nodes, so PageRank does not require atomic accesses. We modified both applications to partition vertices across CPU and GPU cores.

From Chai we use data partitioned and task partitioned applications. Input partitioned histogram (HSTI) is a data partitioned algorithm where CPUs and GPUs use fine-grained synchronization to pop image task blocks from a shared queue and atomically update histogram bins. In-place transposition (TRNS) is a data partitioned matrix transpose algorithm that uses fine-grained CPU-GPU synchronization to arbitrate between threads that are reading and writing conflicting matrix blocks. Random sample consensus (RSCT) is a fine-grained task partitioned algorithm that uses CPU-GPU synchronization to indicate that a sample parameter set produced on the CPU is ready to be consumed on the GPU. CPU cores sparsely read the input matrix and communicate a small amount of data to the GPU, while every GPU core densely reads the same input matrix. Thus, hierarchical sharing is significant. Task queue system histogram (TQH) is a task partitioned algorithm in which the CPU and the GPU use fine-grained synchronization to respectively push to and pop from a set of histogram task queues. Again, CPU cores communicate a relatively small amount of data to GPUs, while GPU cores densely access an input array. However, in TQH each GPU core accesses a different partition of the array, so hierarchical sharing is minimal. GPU cores also use atomic accesses to update a histogram.

Table 5.7 summarizes the communication patterns and execution parameters of each application. Per the Chai categorization, collaborative applications may implement data partitioning or task partitioning. In addition, we classify synchronization between CPU and GPU cores as either fine-

Table 5.7: Collaborative applications communication patterns and execution parameters. CTs = CPU threads. TBs = GPU thread blocks.

Benchmark Suite	**Application**	**Communication Pattern**				**Execution Parameters**
		Partition	**Synch.**	**Sharing**	**Locality**	
Pannotia [38]	BC	data	fine-grain	flat	high	graph: olesnik [48] vertices: 88,263 edges: 243,088 CTs: 8, TBs: 64
	PR	data	coarse-grain	flat	moderate	graph: wing [48] vertices: 62,032 edges: 402,623 CTs: 8, TBs: 8
Chai [53]	HSTI	data	fine-grain	flat	data: low atomic: high	input: 1,572,864 CTs: 4, TBs: 16
	TRNS	data	fine-grain	flat	low	input: 64x4,096 CTs: 8, TBs: 8
	RSCT	task	fine-grain	hierarchical	data: high atomic: low	input: 2,000x5,922x4 CTs: 1, TBs: 16
	TQH	task	fine-grain	hierarchical	data: low atomic: high	input: 80x133x176 CTs: 1, TBs: 32

grained or coarse-grained, and sharing between CPU and GPU cores as either hierarchical (if a core is more likely to share data with other cores of the same type) or flat (if a core is equally likely to share data with both device types). Finally, we specify whether the application exhibits high or low locality in data or atomic/synchronization accesses.

5.4 Evaluation

Figures 5.4 and 5.5 show the execution time and network traffic breakdown of each cache configuration, normalized to HMG, for the synthetic microbenchmarks and collaborative applications, respectively. Network traffic includes all traffic across the mesh network between any two components. It is broken down by request type, and each request category includes the corresponding responses. The Probe network message category represents Inv and RvkO messages. Hbest and Sbest represent the averages of the L1 cache configurations within Hierarchical and Spandex, respectively, that achieve the lowest execution time for each workload. Although existing fixed-protocol caches may be unable to achieve these best-case performance gains for every workload, SBest and HBest illustrate Spandex's ability to support the best possible cache design for a target workload, with an eye towards future caches that may dynamically adapt their coherence strategy.

5.4.1 Synthetic Microbenchmarks

Overall, each microbenchmark highlights the design tradeoffs described in Section 5.3.2, though secondary effects also impact execution time and network traffic. For each microbenchmark, we compare the effects of using a hierarchical vs. flat cache structure, using MESI vs. DeNovo at the CPU, and using GPU coherence vs. DeNovo at the GPU.

Indirection exhibits very little locality and involves high-bandwidth CPU-GPU communication. As a result, hierarchical configurations (HMG, HMD) exhibit significantly higher execution time and network traffic (HMG, HMD) because all CPU-GPU communication is routed through both a shared LLC and an intermediate GPU L2 cache.

Configurations that use DeNovo rather than MESI at the CPU cache exhibit less network traffic because DeNovo only obtains ownership for updated words in a line. When owned data is replaced in the CPU cache or requested by a remote GPU core, DeNovo caches only need to transfer the owned words while MESI caches transfer the full line.

Finally, configurations that use GPU coherence rather than DeNovo for GPU caches (HMG, SMG, SDG) very slightly outperform the corresponding configurations that use DeNovo for GPU caches (HMD, SMD, SDD) because DeNovo obtains ownership for written data, adding latency when that data is accessed by a CPU core. However, most data produced by the GPU is evicted and written back to the LLC before it is next accessed by the CPU, which is why this has such a minimal effect on performance.

ReuseO exhibits more locality and less CPU-GPU communication than **Indirection**, but the costs of hierarchical indirection are still evident in hierarchical MESI configurations (HMG and HMD), which incur greater execution time and network traffic than Spandex configurations. The choice of MESI or DeNovo at the CPU caches has a negligible effect on execution time and network traffic since both obtain ownership for the written data. However, **ReuseO** benefits from using DeNovo in GPU cores (HMD, SMD, SDD) because obtaining ownership for updates enables the GPU caches to exploit locality present in written data. The increased cache reuse leads to greatly reduced network traffic for configurations with DeNovo coherence at the GPU. It also slightly reduces in execution time, although this is minor because GPUs are relatively tolerant to memory latency.

ReuseS performance is largely unaffected by the indirection incurred by hierarchical configurations. Hierarchical MESI is able to avoid indirection overheads here because the intermediate L2 cache obtains Shared permission for reads and is therefore able to satisfy most GPU L1 misses at a similar cost to a flat Spandex LLC.

Performance for Reuse is much more dependent on the choice of protocol for the CPU cache. Configurations that use MESI at CPU caches (HMG, HMD, SMG, SMD) significantly reduce both execution time and network traffic relative to those that use DeNovo (SDG, SDD) due to the fact that DeNovo will self-invalidate the densely read data before it can be reused.

The choice of GPU coherence strategy has little effect on performance here; neither DeNovo nor GPU coherence support Shared state, so they are unable to exploit locality in the dense reads. In addition, the sparse data writes are not on the critical path and whether they obtain ownership or are written through has a negligible effect on subsequent read latency.

Across all microbenchmarks, the optimal Spandex configuration roughly matches or exceeds the optimal hierarchical configuration, reducing execution time and network traffic by an average 18% (max 31%) and 40% (max 69%), respectively. This demonstrates that Spandex can more flexibly adapt to diverse CPU-GPU sharing patterns than a hierarchical MESI-based configuration.

5.4.2 Collaborative Applications

The collaborative applications have a wide range of memory demands. The impact of each design dimension varies per workload, but the most consistent effect is that a flat Spandex LLC improves performance relative to hierarchical MESI.

BC and **PR** are data partitioned graph algorithms with a flat sharing pattern and irregular input-dependent accesses to the graph array. **BC** accesses the graph structure using atomic updates, and these updates exhibit high temporal locality for the input studied. The most important design dimension for **BC** is whether GPU caches use DeNovo or GPU coherence. DeNovo's use of ownership enables GPU caches to exploit the high locality present in atomic accesses, drastically reducing execution time and network traffic relative to a similar configuration using GPU coherence. The performance impact of DeNovo GPU caches for **BC** is greater than in **ReuseO** because **BC** is able to exploit locality in atomic operations in an unbalanced workload, where they are more likely to be on the critical path. **BC**'s performance is not significantly affected by CPU or LLC

protocol choice.

PR accesses the graph structure using data loads (rather than atomics), which exhibit moderate locality. Here the bottleneck is memory throughput, and the most important design dimension is whether flat Spandex or hierarchical MESI is used at the LLC. A Spandex LLC reduces latency and network traffic for GPU read accesses because it avoids an extra level of indirection and uses a simple last-level protocol. Unlike hierarchical MESI, Spandex can handle a GPU ReqV without allocating a cache line in an intermediate cache, without transitioning to a blocking state, without revoking ownership or triggering a write-back from the current owner, and without requiring sharer invalidation the next time that data is written. Similarly, configurations which use a DeNovo CPU perform slightly better than a MESI CPU for this workload because DeNovo reads do not incur blocking states or revoke ownership from the current owner.

For feasibility of simulation, the input sizes chosen for BC and PR mostly fit in the LLC. While a thorough evaluation of input sensitivity is left for future work, the benefits of Spandex can be expected to hold for a much wider range of graph sizes. For workloads with high locality like BC, the efficient ownership offered by Spandex enables cores to exploit cache reuse in the higher cache levels. For workloads with less reuse potential, the flat cache organization enabled by Spandex reduces indirection latency for misses, whether they hit in the LLC or backing memory.

HSTI and **TRNS** are data partitioned applications with limited data locality and frequent synchronization. The choice of LLC protocol has the most significant performance impact for these applications, and using a flat Spandex configuration reduces execution time and network traffic for both. This is largely due to the reduced indirection for data accesses, which have a relatively high miss rate for these workloads. In addition, frequent inter-device synchronization benefits from Spandex's non-blocking ownership transfer. **HSTI** atomics exhibit high spatial locality. As a result, using line granularity MESI rather than DeNovo at the CPU enables improved atomic reuse and slightly improved performance. **TRNS** atomics, on the other hand, exhibit low spatial locality, and word granularity DeNovo ownership is able to offer slightly better performance by avoiding false sharing.

RSCT and **TQH** are task partitioned algorithms which exhibit hierarchical sharing and fine-grained synchronization with low locality. In **RSCT**, all GPU cores access the same shared array. Hierarchical MESI configurations are able to exploit this sharing at the intermediate cache level,

filtering GPU requests, reducing contention at the LLC, and improving CPU performance. Additionally, **RSCT**'s low-locality atomic accesses benefit from DeNovo's word granularity ownership at the CPU and non-blocking ownership transfer at the Spandex LLC. Combining these effects, hierarchical MESI configurations (HMG, HMD) and Spandex configurations with DeNovo coherence at the CPU (HDG, HDD) both offer performance improvements relative to flat Spandex configurations with MESI CPU caches (SMG, SMD).

TQH exhibits some hierarchical sharing in atomic variables, but reads to disparate data sets dominate the access pattern, limiting available reuse or sharing. As with PR, HSTI, and TRNS, flat Spandex configurations reduce TQH execution time and network traffic relative to hierarchical MESI configurations by reducing indirection and blocking states for data accesses with limited locality.

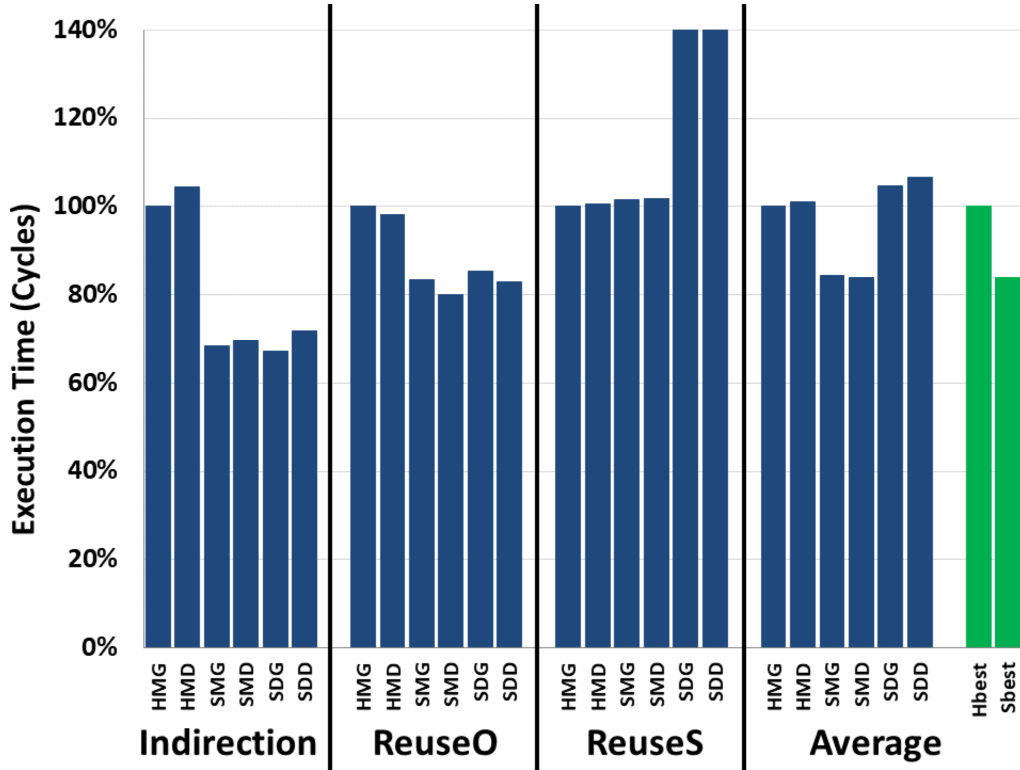
Overall, the benefits of a flat Spandex configuration relative to hierarchical MESI tend to be the most prominent performance effect for the applications studied. Although applications like **RSCT** may benefit from a hierarchical cache structure due to its hierarchical sharing pattern, Spandex can also be made hierarchical to efficiently support such sharing patterns; the reverse is not true of a hierarchical MESI LLC and flat heterogeneous sharing patterns. On average, we find that the best Spandex configuration reduces execution time by 16% (max 29%) and network traffic by 27% (max 58%) relative to the best hierarchical MESI configuration.

5.5 Summary

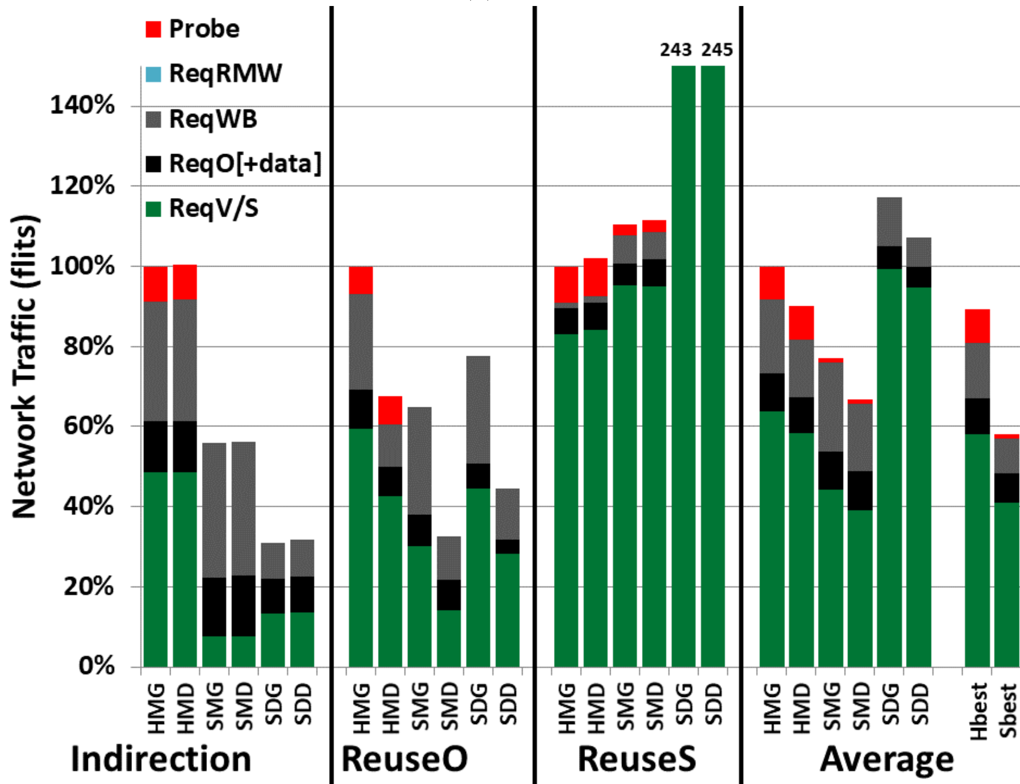
Existing strategies for heterogeneous coherence integration tend to deal with the diverse memory demands of accelerator workloads by adding additional cache layers and/or protocol complexity to an already complex MESI-based protocol. This work defines Spandex, a heterogeneous coherence interface designed to be flexible and simple. By directly interfacing devices that prefer self-invalidated or writer-invalidated loads, owned or write-through stores, and coarse or fine-grained state tracking and communication, Spandex enables CPU, GPU, and other accelerators to efficiently use whichever coherence strategy is most appropriate for the executed workload.

To demonstrate the benefits of Spandex simplicity and flexibility for CPU-GPU coherence, we evaluated it against a less flexible and more complex hierarchical cache structure with a MESI LLC.

We show that an ideally configured Spandex interface is able to reduce execution time and network traffic by on average 16% (max 29%) and 27% (max 58%), respectively, for a diverse range of collaborative CPU-GPU applications when compared with an ideally configured hierarchical MESI coherence interface. These benefits make Spandex a uniquely suitable coherence strategy for the exceedingly diverse workloads of existing, emerging, and future specialized devices.

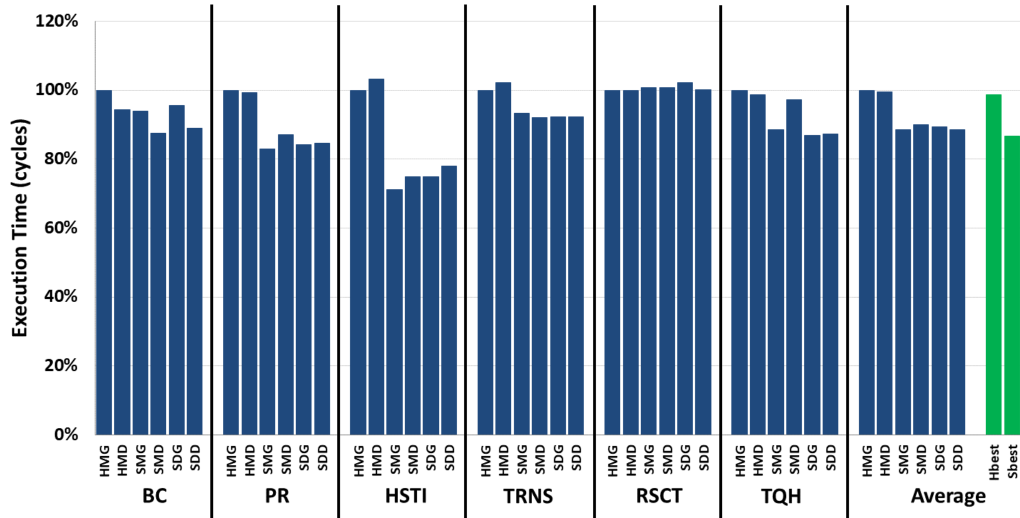


(a) Execution time

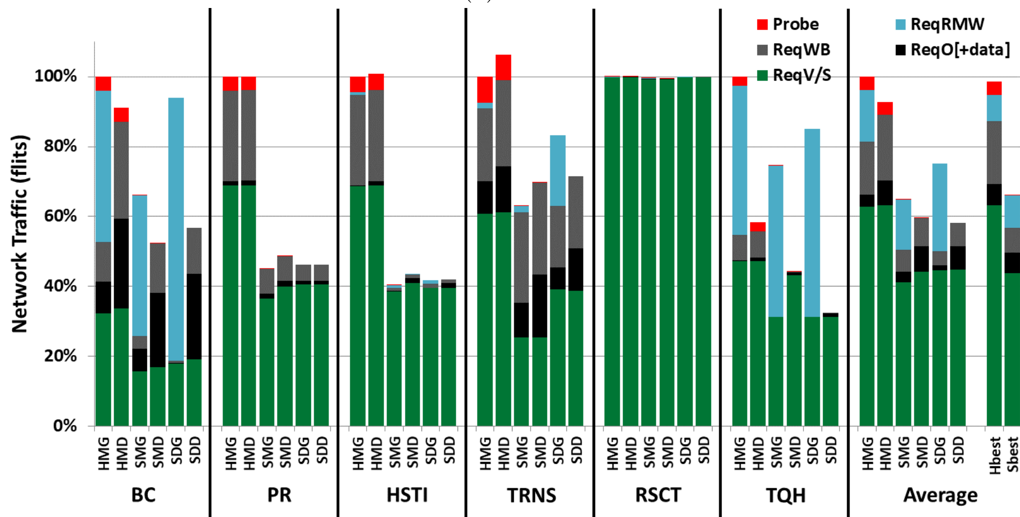


(b) Network traffic

Figure 5.4: Synthetic microbenchmarks execution time and network traffic, normalized to HMG.



(a) Execution time



(b) Network traffic

Figure 5.5: Collaborative applications execution time and network traffic, normalized to HMG.

Chapter 6

Dynamic Coherence Specialization

The flexibility of Spandex offers new opportunities for memory system performance optimization. Although Chapter 5 focuses primarily on Spandex's ability to interface a wide variety of coherence protocols that use fixed request types for each access, there is no need to restrict devices in this way. A local cache in a Spandex system may dynamically choose request types based on the properties of the workload being executed. Spandex requests can also be extended with additional optimizations designed to improve performance for specific access patterns. Unlike MESI-based heterogeneous protocols, which suffer from high complexity and a multitude of blocking states, Spandex is able to implement extensions with minimal effort or added complexity.

Importantly, while this interface flexibility allows a device to choose between multiple types of loads and stores, it does not add complexity to the DRF memory model. No matter which request type is used for a load, store, or atomic access, the functional semantics of that access do not change.

In the following sections we describe a set of specialized Spandex request types, we evaluate the benefits of dynamic request selection effects on a set of applications, and we offer a simple complexity comparison with a more conventional heterogeneous protocol.

6.1 Specialized Coherence Overview

The Spandex protocol offers dynamic flexibility in three dimensions (summarized in table 5.1): stale data invalidation, write propagation, and granularity. Dynamic selection of request types in these dimensions requires no change to the Spandex protocol. We also consider two optimizations that require modification to baseline Spandex: write-through request forwarding and owner prediction.

Dynamic stale data invalidation: Load accesses can generate writer-invalidated reads

(ReqS) if reuse across synchronization boundaries is expected. Since requesting Shared state will prevent requested data from being self-invalidated at synchronization points, this can improve cache efficiency, reducing execution time and network traffic. If reuse across synchronization points is not expected to be high and/or memory throughput is judged to be more important to performance, a load access may generate a self-invalidating read instead and avoid the latency and network overheads of writer-initiated invalidation.

Dynamic write propagation: Stores and atomic operations can use ownership requests (ReqO or ReqO+data) if temporal locality is expected to be high, which can lead to improved cache efficiency, and reduced execution time and network traffic. If stored data will not be accessed again by the writer, and/or the data is expected to be evicted before it is next accessed, write-through requests (ReqWT, ReqWT+data) may be used to avoid the indirection of remote ownership lookup.

Dynamic granularity: Requests may be issued at line granularity if spatial locality is expected to be high (improving cache efficiency, performance, and network traffic), or word granularity if spatial locality is expected to be low (reducing overheads of wasteful data movement).

Write-through forwarding changes how the Spandex LLC handles a write-through request for remotely owned data. Rather than writing the specified data to the LLC, updating the LLC state to Valid, and forwarding an ownership request to the current owner, the write-through request is forwarded to the current owner without updating any state at the LLC. This is similar to how a self-invalidated read (ReqV) request is handled, since neither a forwarded write-through request nor a ReqV updates state at the LLC. As a result, a forwarded write-through request can similarly fail if ownership changes before the request arrives at the owner. When this happens, the failed request must be Nacked and retried. In addition, after some finite number of failures, a requestor must retry with a non-forwarded write-through request to prevent starvation in the presence of frequent ownership changes.

Write-through forwarding can be useful when write locality is low and the next consumer of written data is expected to currently own it. From a programmer's perspective, a consumer may request ownership for some data in a buffer as a kind of pre-loading step, or if a space is repeatedly used for message passing the consumer can obtain ownership once on-demand and hit locally for subsequent accesses. Subsequently, the producer may use forwarded write-through requests to send

the data directly to the consumer's cache, enabling it to hit on subsequent read accesses. Since read latency is typically more important to performance than write latency, moving the overhead of data movement to the producer side is often beneficial.

Owner prediction takes advantage of Spandex support for self-invalidated reads and forwarded write-through requests to allow a device to predict the current owner of target data and send a request directly to that core. This is identical to the direct cache-to-cache transfer optimization described by Choi et al. for self-invalidated reads in the original DeNovo protocol [42], but with Spandex it can be applied to write-through requests as well. Neither request type affects state at the LLC; when the target data is remotely owned, the request is simply forwarded to the current owner. This means that sending such a request directly to the current owner is functionally equivalent to looking up the current owner at the LLC. In addition, since these forwarded messages are not serialized with other requests in Spandex (which may change the ownership state), and since Spandex assumes an unordered network, the protocol must already account for forwarded ReqV or ReqWT requests arriving after ownership has been lost, prompting a retry. Therefore, mispredicting an owner may have negative performance effects, but it will not break anything in the protocol.

If the current owner is known or can be predicted, using owner prediction to directly forward ReqV or ReqWT requests avoids the latency, traffic, and energy overheads of looking up the owner at the LLC. If the physical location of the owner device can be predicted at compile time (e.g. there is a dataflow edge between fixed devices), this information may be included as a software hint. Otherwise, a lightweight predictor could be used to guess the current owner based on past responses to forwarded ReqV or ReqWT requests.

Determining how best to use these specialized coherence request types is a significant challenge in itself, requiring knowledge about the memory demands of the workload. Although outside the scope of this work, past research has focused on extracting this type of information statically from software, at compile time, or dynamically at runtime. For example, partitioned global address space (PGAS) programming models carry explicit information about expected data access locality, visibility, and throughput demands that are used to guide the layout of data in a distributed memory system [121, 87, 35, 37, 50, 120, 36, 20], scoped synchronization accesses in the HSA program-

ming model carry information about synchronization locality [51], [1] and dataflow programming languages offer explicit information about producer-consumer relationships and data movement granularity [122, 1]. Sometimes locality and dataflow information is not statically available, or the optimal request types depend on the resources of the target system or how tasks are dynamically scheduled in a system. In these cases, request types may be selected based on compile-time analysis or runtime profiling.

6.2 Complexity Analysis

Two of Spandex's biggest strengths are flexibility and simplicity: it offers a wide range of access types to fit any workload's memory demands while avoiding much of the complexity that makes modern heterogeneous protocols so difficult to validate or extend. The goals of flexibility and simplicity are often at odds with each other, and Spandex is able to achieve both by building upon the simple and efficient DeNovo protocol. DeNovo relies on word granularity state and a DRF memory model to entirely avoid the need for transient blocking states. Komuravelli and Adve used the Murϕ model checker tool [49] to compare the complexity of the DeNovo and MESI protocols and found MESI to be significantly more complex; the Murϕ state space exploration explored 14x more states [62].

Spandex extends DeNovo to support more request types, such as writer-invalidated reads and write-through stores and atomics. This added flexibility introduces transient states and new race conditions, increasing the state space significantly. However, since it is based on DeNovo rather than MESI, we claim it is still preferable to more conventional heterogeneous coherence solutions.

To evaluate this claim, we explore the DeNovo-based Spandex protocol state space in Murϕ and compare it against a MESI protocol (based on the implementation used by Multifacet GEMS [82]) extended to implement a simplified version of non-snoopable request types similar to those of the ARM CHI [16], described in Section 5.1.2 with added support for accessing any data with any request type. A flat cache organization was used instead of a hierarchical heterogeneous protocol such as that of the gem5 AMD APU [13] because exhaustively exploring interactions between all devices in a multi-level protocol is expected to require a prohibitively large state space. In addition,

[1] Although using the HRF memory model incurs undesirable complexity, we assume scopes may be used as optional hints to communicate expected locality in a DRF memory model.

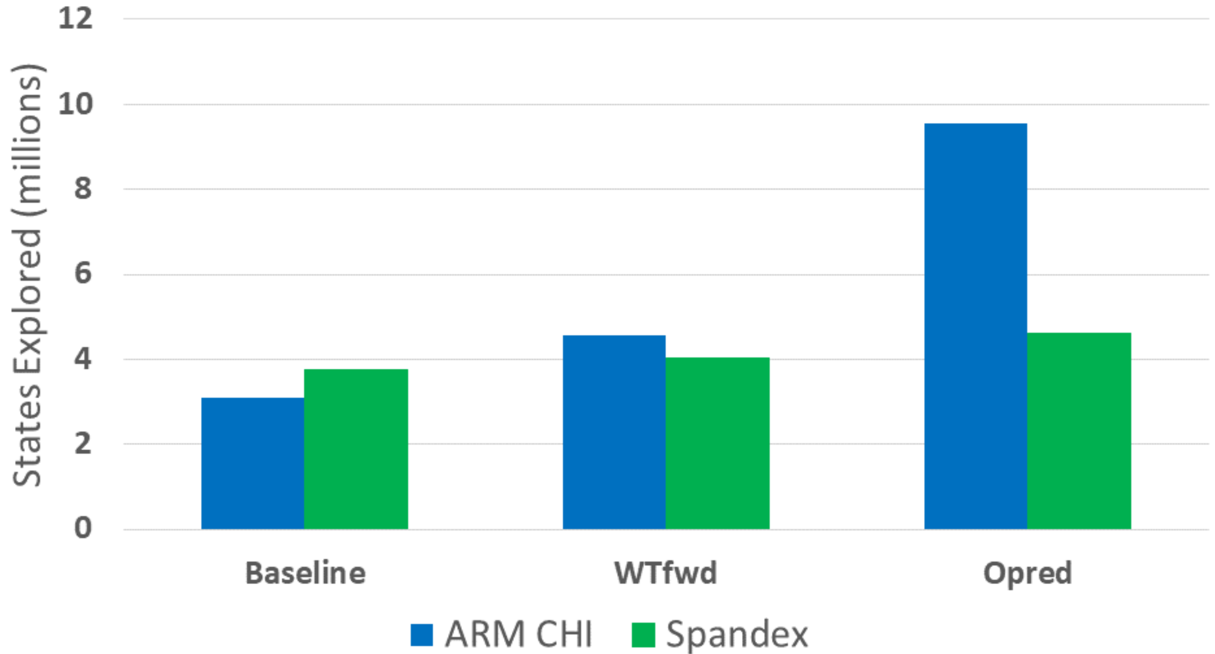


Figure 6.1: State space comparison between Spandex and ARM CHI with added optimizations, writethrough forwarding (WTfwd) and owner prediction (Opred).

although Spandex supports variable request granularity, we do not explicitly model this in Murϕ and instead assume a fixed word granularity. A Spandex system is expected to handle each word in a multi-word request individually based on its state. Therefore, we contend that adding support for coarse-grained or multi-word request types does not add any fundamental complexity to the protocol and would unfairly increase the state space in a Murϕ model.

Each of these protocols is then further extended with the write-through forwarding and owner prediction optimizations discussed in Section 6.1. The actual CHI specification in fact supports a form of producer-consumer forwarding, referred to as cache stashing, and cache stashing differs slightly from the optimizations described above (this difference is discussed in Section 7.4 in more detail). However, the above optimizations are applied identically to both designs to ensure a fair comparison.

Figure 6.1 plots the number of states explored by Murϕ for each protocol model. Interestingly, the baseline Spandex protocol actually has slightly more states than the baseline ARM CHI model. This can be attributed to the fact that Spandex offers fundamentally more flexibility. Rather than simply two types of load and store requests (snoopable and non-snoopable), Spandex allows the programmer to specify whether or not a store requires up-to-date data as well, and thus requires

considering two more L1-initiated request types that may be pending at the L1 or L2 at any time. This added request type flexibility enables devices in a Spandex system to avoid much of the false sharing and blocking states that are unavoidable in CHI, giving the developer improved control over some of the overheads associated with protocol complexity.

As optimizations are added, the fundamental complexity of the MESI-based CHI coherence solution becomes more evident. Murϕ explores 1.1x more states in ARM CHI with write-through forwarding (WTfwd) added to both protocols, and 2.1x more states with owner prediction (Opred) added. This additional complexity can be explained by the fact that CHI is based on a line-granularity read-for-ownership protocol, and this prioritization is reflected in the state transitions. Every cache miss causes a transition to a transient state that blocks subsequent requests until some satisfying set of messages is received to transition back to a stable state. In contrast, Spandex builds upon DeNovo (which has no transient states) and avoids blocking states as much as possible (they are only needed for transitioning to and from Shared state at the LLC, and for supporting ReqWT+data requests). Spandex therefore avoids much of the state space explosion that can occur when adding optimizations to a protocol with excessive transient states. In fact, there is little increase at all in the number of Spandex states.

6.3 Evaluation

6.3.1 Methodology

To evaluate the above dynamic coherence flexibility, we use an integrated CPU-GPU architectural simulator with six architectural configurations. The simulator is identical to the simulator used in Section 5.4 with added support for dynamic request selection and write-through forwarding. The first four configurations (SMG, SMD, SDG, SDD) are the same as the Spandex configurations in Chapter 5.4. Optimized Spandex (SO) uses selective dynamic request selection and (if beneficial) write-through forwarding to optimize memory system performance. For this evaluation, the optimal memory request type is manually selected based on the known or profiled properties of the workload. SO+pred adds owner prediction to SO, which means ReqV and ReqWT requests are sent directly to the current owner (modeled as an oracle prediction, using the current state at the LLC).

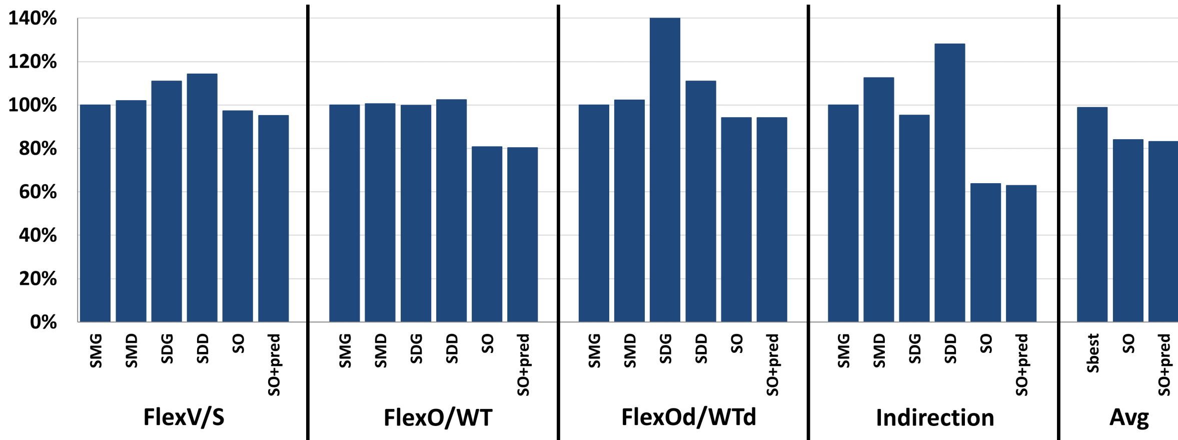
We compare these configurations using a set of synthetic microbenchmarks which highlight the

value of these memory optimizations. These benchmarks and the manually chosen request types for each are described next.

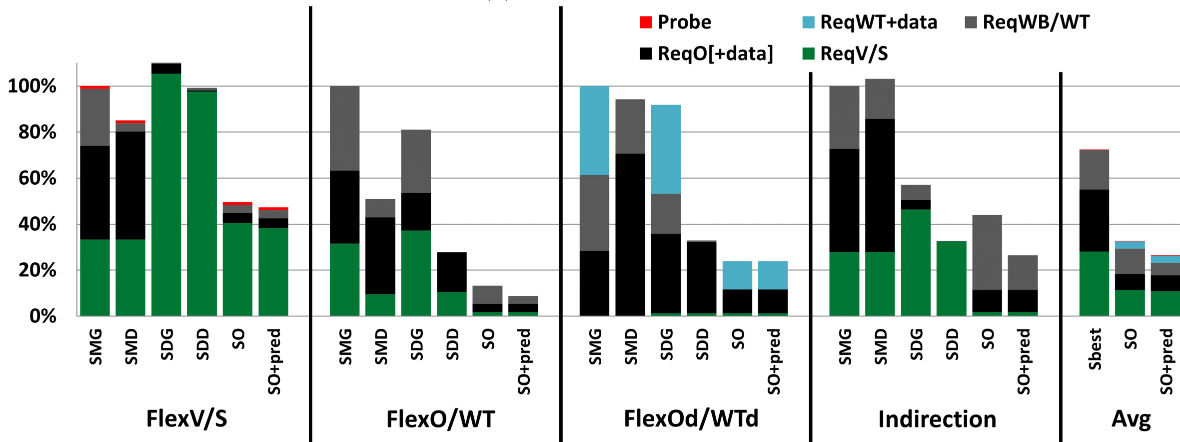
- **FlexV/S**: Highlights the benefits of dynamically choosing between writer-invalidated and self-invalidated reads. CPU and GPU cores repeatedly access tiles in multiple arrays, and only some accesses exhibit inter-kernel reuse. The CPU densely reads one array with high inter-kernel reuse (array A) and with overlap with other CPUs, then densely reads a second array with no inter-kernel reuse (array B). The GPU sparsely stores to array A with no inter-kernel reuse, and densely reads and writes array B with high inter-kernel reuse. In optimized Spandex, CPU loads to array A use writer-invalidated reads, while CPU loads to array B use self-invalidated reads. GPU updates to array A use write-through requests, while GPU updates to array B obtain ownership.

- **FlexO/WT**: This microbenchmark highlights the benefits of dynamically choosing between ownership-based stores and write-through stores. It is similar to the ReqO microbenchmark from Section 5.4 where CPU and GPU cores take turns densely reading and writing one matrix with high inter-kernel reuse and sparsely reading a second matrix with low inter-kernel reuse. However, in FlexO/WT the sparsely accessed matrix is written instead of read. This sparse update exhibits low inter-kernel locality, so obtaining ownership is wasteful. Optimized Spandex uses line granularity ownership requests for the dense writes with high locality and uses word granularity forwarded write-through requests for the sparse updates with low locality.

- **FlexOd/WTd**: This microbenchmark highlights the benefits of dynamically choosing between ownership-based atomic accesses and write-through atomic accesses, as well as dynamic granularity selection. Each core (CPU and GPU) atomically updates one matrix densely with high reuse potential and one matrix sparsely with low reuse potential (and in a racy manner). The densely accessed matrix is also accessed sparsely by a remote core concurrently (motivating the atomic request type). Optimized Spandex uses line-granularity ownership requests (ReqO+data) for atomic accesses with high reuse potential and uses word-granularity write-through requests (ReqWT+data) for atomic accesses with low reuse potential.



(a) Execution time



(b) Network traffic

Figure 6.2: Synthetic microbenchmarks execution time and network traffic, normalized to SMG.

- **Indirection**: This is the same as microbenchmark in Section 5.4, where CPU and GPU cores take turns reading a tile from one matrix and writing it to the transposed tile in another matrix. Optimized Spandex obtains ownership for reads and uses forwarded write-through requests for the writes.

6.3.2 Results

Figure 6.2 shows the execution time and network traffic for the microbenchmarks. Network traffic is broken down by request type as in Sections 3.2.3 and 5.4. Sbest represents the best performing (lowest execution time) non-optimized Spandex configurations for each microbenchmark averaged over all microbenchmarks.

For **FlexV/S**, the best performing non-optimized configurations are SMG and SMD, so the use of writer-invalidated reads at the CPU is clearly important to performance. Obtaining Shared state for the dense read accesses with inter-kernel reuse improves CPU cache hit rates, reducing latency and traffic for these CPU read accesses. Obtaining Shared state for the sparse read accesses with no inter-kernel reuse is wasteful, and requires revoking ownership (when enabled) from a remote GPU core. Revoking this ownership also prevents inter-kernel reuse at the GPU. Optimized Spandex requests line granularity Shared state for the dense reads with inter-kernel reuse and it uses word granularity self-invalidated reads for the sparse accesses with no reuse. It therefore offers high cache hit rates for the data with high locality while avoiding the overheads of Shared state where it is wasteful (and allowing the GPU to exploit reuse in owned data). SO as a result reduces network traffic by 50% relative to the fastest non-optimized configuration, SMG (42% relative to the configuration with lowest network traffic, SMD). However, performance is dominated by the latency of the CPU accesses, which exhibit high potential for reuse, and SO's ability to avoid wasteful shared state and improve GPU reuse only decreases total execution time by 3% relative to SMG (GPU performance is less sensitive to memory latency). With ownership prediction (SO+pred), avoiding the indirection of lookups at the LLC for ReqV and ReqWT requests does not significantly improve performance, reducing network traffic only by an additional 5% and execution time an additional 2% relative to SO. This is because forwarded ReqV requests do not carry data, so they do not contribute significantly to total traffic. In addition, owner prediction can only change a three-hop miss to a two-hop miss. This latency difference is relatively small, and it only affects a subset of CPU accesses, reducing the impact of the optimization.

For **FlexO/WT**, execution time is relatively constant among non-optimized configurations, but network traffic varies greatly. This workload exhibits significant inter-kernel reuse in written data, so obtaining ownership for writes is important. Since overall performance is most sensitive to CPU memory latency and all configurations obtain ownership for writes at the CPU, there is not much difference in execution time. On the GPU, however, only SMD and SDD obtain ownership for writes, and the improved GPU cache hit rate reduces network traffic by up to 65% among non-optimized Spandex configurations (SDD vs. SDG). FlexO/WT also contains sparse stores to remotely owned data, which exhibit no inter-kernel reuse and require revoking ownership from the core that experiences reuse in unoptimized Spandex. Optimized Spandex uses forwarded write-

through requests for these stores, performing the updates in place at the current owner and avoiding a wasteful ownership revoke. As a result, Spandex increases the load hit rate for both CPU and GPU caches, reducing execution time by 19% relative to the fastest non-optimized configuration (SMG) and network traffic by 87% relative to the non-optimized configuration with the lowest network traffic (SDD). With oracle ownership prediction, it is possible to reduce network traffic by an additional 34% relative to SO.

For **FlexOd/WTd**, execution time and network traffic for non-optimized Spandex configurations are relatively constant apart from two outliers: exceptionally high execution time for SDG, and exceptionally low network traffic for SDD. These differences are due to differences in the type of access used for atomics. At the CPU, both DeNovo and MESI use ownership for atomics, but DeNovo uses word granularity while MESI uses line granularity. Both are able to exploit reuse in the dense accesses with high inter-kernel locality, but for sparse accesses with no locality, MESI incurs overheads of false sharing by requesting ownership at line granularity. This is seen in the high network traffic for SMG and SMD. Interestingly, this false sharing overhead is also present in SDG. The reason for this has to do with how Spandex handles ReqWT+data requests for remotely owned data. When a ReqWT+data request from the GPU arrives for data owned at the CPU, ownership is revoked at line granularity (this is done for simplicity and is not required for functionality). In a MESI CPU, the first atomic access to the line in the subsequent iteration will retrieve ownership for the full line, exploiting spatial locality in dense atomic accesses. However, DeNovo will request ownership for each word individually. This is what contributes to the significant performance degradation for SDG. In contrast, with word granularity ownership requests at both the CPU and GPU, SDD is able to exploit reuse while avoiding the network traffic overheads of false sharing. Optimized Spandex is able to treat atomic accesses individually: it uses word granularity write-through requests for sparse accesses with low locality, and line-granularity ownership requests for dense accesses with high locality. For the sparse racy accesses to remote arrays, optimized Spandex uses word granularity ReqO+data to avoid revoking ownership at line granularity (as happens with SDG). SO therefore exploits cache reuse where possible and avoids wasteful data movement when locality is not present. It is able to reduce execution time even more than the fastest unoptimized configuration (6% reduction relative to SMG), and reduce network traffice even more than the unoptimized configuration with the least traffic (27% reduction relative

to SDD). Owner prediction provides no benefit here since only atomic requests are used (only ReqV and ReqWT requests can be directly forwarded).

In **Indirection**, execution time is most affected by the choice of GPU protocol, while network traffic is most affected by the choice of CPU protocol. Using DeNovo rather than GPU coherence at the GPU increases execution time because CPU cores read data written at the GPU. Although obtaining ownership for writes at the GPU can reduce network traffic by avoiding data copies through the LLC, it adds indirection latency for reads at the CPU, increasing overall execution time. Using MESI at the CPU can increase network traffic primarily due to false sharing. In this tiled matrix transpose workload, loads exhibit high spatial locality, but multiple cores may store to a single cache line. Using line-granularity ownership requests requires ownership transfer and data movement even when stores are to disparate words. Optimized spandex avoids all of these overheads and is actually able to exploit reuse in reads despite the fact that read values are updated remotely between each kernel invocation. It does this by obtaining ownership for loads, using write-through requests for stores, and using ownership prediction to send stores directly to the current owner. By moving the overhead of data movement from the reader to the writer side, SO improves read hit rates, reducing execution time by 33% relative to the fastest non-optimized configuration (SDG). However, indirection also highlights one side effect of write-through forwarding: forwarded write-through requests incur additional traffic. This is because update data must travel two hops: first to the LLC and then to the owning core. In contrast, configurations that obtain ownership for updates only need to transfer data one hop: from the owner to the requestor (all other hops do not transfer data). As a result, SO increases network traffic by 35% relative to SDD (the unoptimized configuration with the lowest traffic). However, this overhead can be avoided using ownership prediction, and SO+pred reduces network traffic by 54% relative to SDD.

Overall, the flexibility offered by dynamic request selection, write-through forwarding, and owner prediction enables notable performance gains for the workloads studied. SO (SO+pred) reduce execution time by on average 15% (16%) and network traffic by on average 55% (63%) relative to the fastest non-optimized Spandex configuration for the microbenchmarks studied. Although not explored here, we intend to investigate the benefits of dynamic coherence specialization in real heterogeneous applications in future work.

6.4 Summary

As computer systems become more parallel and heterogeneous, flexible memory systems like Spandex are needed to support their increasingly diverse memory demands. The flexibility and simplicity of Spandex allow devices to dynamically adapt their coherence strategies to exploit predicted or statically known workload properties. Spandex offers request type flexibility in the fundamental coherence dimensions of stale data invalidation, write propagation, and request granularity. It is based on the simple DeNovo protocol, and this allows it to incorporate extensions and optimizations more easily than more complex MESI-based heterogeneous coherence solutions.

In this work we describe dynamic coherence specialization, additions to the memory request interface that enable software or a hardware predictor to tailor a coherence request type to the access properties of the executing workload. While this additional information does not affect the functional semantics of memory accesses, we show that it can be used to improve memory system performance for a range of access patterns. In addition, using the Murϕ model checker tool, we show that although baseline Spandex is slightly more complex than a conventional MESI-based heterogeneous protocol, it is more flexible and easier to extend with optimizations. Going forward, it will be important to examine the memory demands of future workloads to determine which types of coherence optimizations can offer the greatest benefits.

Chapter 7

Related Work

There have been many past efforts to address the challenges associated with designing memory systems for emerging heterogeneous devices. In this chapter we describe many of these efforts and discuss how our contributions differ from, build upon the insights of, or can be applied in conjunction with these prior innovations. We divide related work into four broad areas: 1) performance evaluation for high-throughput accelerators (work related to Chapter 2), 2) efficient heterogeneous coherence (work related to Chapters 3 and 5), 3) atomic relaxation in heterogeneous systems (work related to Chapter 4), and 4) dynamic coherence specialization (work related to Chapter 6).

7.1 Performance Evaluation for High-Throughput Accelerators

There have been previous efforts to provide more detailed performance profiling metrics for high-throughput accelerators such as GPUs. GPGPU-Sim is a cycle-accurate GPU simulator that can be used to model the effects of architectural changes to a GPU device [17]. It provides many simulation evaluation metrics including overall performance and stall counts, but not a stall breakdown. Aerialvision [15] augments GPGPU-Sim to provide more detailed measurements such as the number of cycles spent in the core pipeline by each thread, the number of DRAM accesses generated by each instruction, and the number of stall cycles due to long-latency off-chip read requests. However, it lacks a single comprehensive breakdown attributing execution latency to fine-grained stall causes.

O'Neil and Burtscher [90] use a stall classification method, also based on GPGPU-Sim, to help understand the performance of irregular GPU kernels. Their breakdown, however, does not differentiate between stalls due to memory or compute delays. Lee and Wu [72] also use a similar GPU stall classification method based on GPGPU-Sim which is designed to characterize the latency-hiding capability of GPU warp schedulers. It provides a more detailed stall breakdown and adds a

memory latency stall type, but is focused on warp scheduler changes.

While this prior work also provides profiling information about GPUs, unlike GSI, they are all designed for discrete GPUs. Furthermore, they focus on profiling the GPU core, while we are capturing detailed information about the causes of memory stalls in a unified CPU-GPU memory system.

7.2 Efficient Heterogeneous Coherence and Coherence Integration

Many prior innovations aim to both improve heterogeneous coherence and integrate heterogeneous caches. We therefore discuss work related to the contributions in Chapters 3 and 5 together. We omit discussion of HRF and RSP, which are described in Section 3.1, as well as IBM CAPI, AMD APU, and ARM CHI, which are described in Section 5.1.

There are multiple ongoing industry efforts to define communication interfaces between devices in heterogeneous systems. For example, CCIX [56], Gen-Z [43], and OpenCAPI [45] present interfaces for performing cached or non-cached coherent read and write accesses in a heterogeneous system. These interfaces are more expressive than Spandex in some respects. For example, by offering an interface for attached devices to act as the home memory node for an address range or by offering bulk cooperative synchronization transactions such as reduce and gather/scatter operations. They also offer more low-level flexibility regarding routing, packet loss and error correction/detection, and general physical layer configuration options. However, at the time of writing, the underlying coherence protocol and state transition internals at the host/responder/home agent (in OpenCAPI/Gen-Z/CCIX terminology, respectively) for these interfaces are not publicly available. Marvell MoChi is a modular chiplet architecture that promises seamless integration between components [44], but again, a detailed specification of any coherence interface between these chips is not publicly available. The fact that there are so many active industry efforts to address this challenge serves to reinforce its growing importance. Though many implementation details are still not public or are under development for these integration strategies, the insights presented in this work may serve to motivate a focus on simplicity and flexibility in future iterations.

Past work has implemented efficient CPU-GPU coherence, using both hardware and software

techniques. QuickRelease (QR) [54], Heterogeneous System Coherence (HSC) [95], and the Fusion architecture [67] all use a clustered hierarchical cache structure and MESI-based last level directory to interface CPU and GPU devices. Software-managed page migration strategies that intelligently perform explicit page copies between CPU and GPU memories have also been shown to be effective for many CPU-GPU workloads [86, 97, 8]. However, all of these techniques rely on dense, hierarchical sharing patterns to be effective, and performance can suffer for new or irregular sharing patterns due to the high cost of inter-device communication.

Interfacing heterogeneous protocols can be avoided by using the same protocol at both CPU and GPU L1 caches. Temporal coherence [103], VIPs-G [66], and even DeNovo for GPUs [100] adopt this approach. However, the design of an L1 interface is often tightly integrated with processor design, and requiring all cores to interface with a new cache protocol may be an unacceptable design burden.

Crossing Guard offers a simple and stable coherence interface for accelerator caches [89]. However, Crossing Guard's primary goal is correctness and security in heterogeneous coherence, not improved performance. It also only interfaces with MESI and MOESI style caches. Thus it is unclear whether accelerator caches that prefer simpler protocols such as GPU coherence or DeNovo would be able to efficiently interface with the system.

Manager-client pairing (MCP) is a framework for defining hierarchical coherence protocols in large-scale systems [22]. Like Spandex, MCP addresses the complexity of hierarchical protocols and evaluates the costs of deeper cache organizations. However, MCP is focused on providing a generic framework for building hierarchical protocols for devices that request read and write permission. Spandex instead aims to avoid the need for hierarchical organization for a wide variety of heterogeneous devices, some of which use self-invalidations or write-through caches instead of requesting read/write permissions, and which can use variable request granularity.

Past work has also addressed the challenge of enforcing memory consistency in heterogeneous systems. ArMOR provides a framework for precisely defining the ordering requirements of different consistency models and ensures that devices with different memory models respect ordering constraints in a heterogeneous system [77]. The Princeton Check family of tools offer a framework for specifying and verifying memory consistency ordering constraints, considering microarchitectural through application level concerns in the design of a coherent device [115, 79, 77, 76, 80, 75]. These

techniques may be used in conjunction with Spandex when designing and integrating new device types.

7.3 Atomic Relaxation in Heterogeneous Systems

Many existing memory models attempt to formalize allowed behavior in the presence of relaxed atomics while also prohibiting the strange behavior that can arise from out-of-thin-air (OOTA). Boehm and Demsky thoroughly describe the problem of OOTA and propose a solution - preserving load-store ordering - which would prohibit common existing performance optimizations which exist in current hardware [29]. The HSA memory model conservatively prohibits dependency cycles, preventing some desirable optimizations [51]. The Java memory model [81] defines complex semantics which aim to enable as many optimizations as possible, but has been found to unintentionally restrict some very common ones [99], and no fix has yet been proposed. The C++11 memory model also was found to unintentionally restrict certain executions [28, 26], and C++14 simply informally states that systems should not produce out-of-thin-air values [27].

Other attempts at formalization rely on difficult to understand concepts such as event structures and promise semantics [93, 58]. In summary, existing solutions to the challenge of relaxed atomics are either too ambiguous or too complex, and none preserve the SC-centric semantics used by DRFrlx.

7.4 Dynamic Coherence Specialization

Adaptive coherence strategies have been explored extensively in prior work to improve memory system performance. Dynamic self invalidation (DSI) offers adaptivity in the form of tear-off blocks, which use self-invalidation to avoid the need for tracking sharers or sending invalidations for read data, effectively enabling dynamic selection of self-invalidated or writer-invalidated reads [71]. Last-touch prediction can be used to avoid the indirection of a remote read request for modified data by selectively writing back modified data to the directory when an instruction is expected to be the "last touch" to that data [70]. It can achieve benefits similar to those obtained by dynamically selecting write-through or ownership-based stores. Past work has also studied the benefits of flexible state or coherence granularity [114, 123, 34, 96, 68, 47], which show that adapting communication

and state tracking granularity to the spatial locality of a program can reduce false sharing and enable more efficient data tracking, lookup, and transfer. Finally, protocol extensions have been proposed which detect and predict specific sharing patterns, optimizing requests to reduce wasteful traffic and latency for those patterns [105, 46, 65, 14, 2, 85, 59, 24, 83, 41, 104, 69]. All of the above innovations demonstrate the benefits of dynamic adaptability, and in some cases offer even more flexible control over granularity or data movement than Spandex request types. The methods for detecting and predicting sharing patterns will be increasingly relevant to specialized memory systems, and they could be used to help guide Spandex request type selection. Even so, all of the above innovations focus on optimizing communication patterns in the context of MESI-based multicore CPU coherence. As a result, their flexibility is limited, each focusing on adaptivity within a single coherence design dimension, and the benefits they provide often come at the cost of additional overhead and complexity on top of an already complex MESI-based protocol. In contrast, Spandex is designed to offer comprehensive coherence flexibility while retaining a simple protocol with minimal coherence overheads.

Many studies also have proposed ways to improve memory access efficiency through adaptive cache management (replacement, prefetching, bypassing) in both CPUs and GPUs based on hints from the software, compiler, or runtime profilers [118, 23, 119, 92, 98, 40, 113, 74, 64]. These techniques allow caches to adapt to different access patterns and improve cache efficiency, and many of the insights regarding cache policy prediction can be leveraged to help select Spandex request types. However, their flexibility is constrained to cache management policies rather than the fundamental coherence protocol flexibility offered by Spandex.

Expressive memory was recently proposed by Vijaykumar et al. as a framework for communicating high level properties of a program's memory access pattern to the underlying hardware to guide architectural optimizations, primarily in multicore CPUs [117]. Similarly, the locality descriptor communicates high level information about data access locality in GPU programs to guide thread block scheduling and data placement in GPUs [116]. Both of these innovations exploit high level information from software to adaptively optimize memory system performance. These techniques are entirely compatible with a Spandex system, and they could be used to take advantage of dynamic coherence specialization by guiding request type selection.

The ARM CHI interface supports a mechanism for producer-consumer forwarding called cache

stashing. This is similar to the write-through forwarding and owner prediction optimizations described in Chapter 6 in that a producer device provides a hint to the system regarding the expected location of the consumer of some data. However, unlike the write-through forwarding and owner prediction optimizations, cache stashing causes exclusive ownership to transfer to the target cache. Although this avoids the need for the consumer to pre-emptively obtain ownership, it incurs additional complexity in the form of transient states, and it prevents direct device-to-device communication since a stash request may change the owner state at the home node.

7.5 Summary

A large body of past work has focused on challenges similar to those described in this thesis. Prior methods for high-throughput performance characterization offered valuable insights for discrete GPU architectures, but limited information about memory stalls in a tightly coupled heterogeneous memory system. Many strategies have been proposed for implementing coherence for high-throughput devices, but they often achieve performance gains at the cost of a complex protocol or memory consistency model. Relaxed atomics has been thoroughly studied in past work, but there has been no acceptable SC-centric solution, even as their importance grows with the rise of heterogeneous computing. There are also many past efforts to dynamically optimize coherence protocols for certain access patterns. Often these innovations assume a MESI-based protocol, or can motivate, complement, or be leveraged to select the optimized request types described in Chapter 6. Overall, the contributions of this thesis build upon past work to enable a memory system that is in many ways simpler and more flexible than what has previously been proposed in an age of heterogeneous computing.

Chapter 8

Conclusion and Future Work

8.1 Summary of Thesis

With the end of Dennard scaling, specialized hardware has become an important driver of performance in many compute domains. To flexibly support a broad range of workloads, emerging heterogeneous systems require a comprehensive coherence strategy to enable efficient synchronization and communication between these diverse devices. Implementing heterogeneous coherence requires simultaneously 1) developing and evaluating coherence strategies which are appropriate for each individual device, and 2) implementing efficient coherence between devices which use different strategies. The first challenge is especially difficult for high-throughput devices such as GPUs. Conventional coherence strategies suffer from inefficient synchronization, and recent efforts to address this weakness add significant complexity to the memory model. Existing solutions to the second challenge tend to suffer from high complexity as well, adding request types and blocking states to an already complicated MESI-based protocol designed for CPUs.

In this thesis, we show that it is possible to enable efficient coherence for a diverse range of specialized systems without resorting to complex consistency models or coherence interfaces. First, with GPU Stall Inspector we offer a framework for evaluating the performance effects of coherence innovations in a highly parallel device. Next, we present two such coherence innovations - DeNovo for GPUs and Heterogeneous Lazy Release Consistency - which offer significantly improved performance for emerging GPU applications without adding complexity to the DRF memory model. With DRFrlx, we formalize safe uses of relaxed atomics, and we demonstrate how they can enable significant performance benefits for heterogeneous workloads while retaining an SC-centric memory model. To address the challenge of heterogeneous coherence integration, we introduce the Spandex coherence interface, which offers improved flexibility, simplicity and performance when compared

with conventional methods for coherence integration. Finally, we show how Spandex's flexibility and simplicity can be exploited and extended to further improve performance in emerging specialized systems.

8.2 Future Work and Impact

With the above contributions we show that unified coherent memory is possible for specialized systems without requiring complex coherence or consistency strategies, and together they open the door to numerous potential research directions, some nearer term and some longer term.

8.2.1 Near-Term Research

In the near term, there are many opportunities for refining and optimizing the innovations described in previous chapters for even more application domains. Given the limitations of quantum atomics in the DRFrlx memory model, atomic types and semantics could be added to the model which make new relaxed patterns more feasible. For example, in some types of pointer-based synchronization constructs relaxation with DRFrlx is still difficult. A quantum load to a pointer value may return any value in a quantum-equivalent execution, and it may be beneficial to further limit the possible return values to prevent races when processing and performing data accesses based on this value.

Spandex makes it feasible to integrate a much broader range of devices into a coherent heterogeneous system, and exploring how devices such as FPGAs and ASICs interact with the Spandex interface and with each other could help guide further coherence optimizations. Integrating optimizations such as hLRC, scopes for locality hints, and DeNovo regions into the Spandex implementation could also be used to improve performance. In addition, the principles of Spandex could be extended to multi-chip environments. This would introduce a new set of challenges and trade-offs surrounding the importance of home node data placement and the longer latency inter-chip hops that come with non-uniform memory access (NUMA) systems. However, we expect the flexibility, simplicity, and scalability benefits of Spandex make it uniquely suited to implementing cross-chip coherence. For example, the cost-efficient ownership and easy extensibility of Spandex could enable nodes to dynamically choose whether to locally cache data from a remote home node in a way that avoids many of the undesirable overheads of multi-chip coherence tracking.

The benefits of dynamic coherence specialization could also be further explored, and it would be interesting to investigate how they affect performance for a set of real world applications. Automating Spandex request type selection (by predicting sharing properties with a compiler or hardware predictor rather than hard-coding them) would also improve the practicality and usability of dynamic coherence specialization.

8.2.2 Long-Term Research

In the longer term, there are many opportunities to broaden the scope of the specialized coherence requests described in Chapter 6. Dynamic request selection, write-through forwarding, and owner prediction only scratch the surface when it comes to increasing specialization and flexibility in emerging memory systems. The information available from emerging programming languages, compilation tools, and runtime profiling techniques can be used to exploit not only the flexibility offered by the Spandex coherence interface, but also that offered by entirely novel architectural coherence innovations. Two such examples, coherent hardware queues and efficient coherent DMA, are described next, followed by a summary of the expected future impact of the innovations in this thesis.

Coherent hardware queues: The queue data structure is a valuable construct for asynchronous communication between multiple threads or devices, and queues may be implemented coherently in software or as a hardwired communication path in hardware. The goal of coherent hardware queues is to simultaneously achieve the global addressability and programmability of coherent software queues while also offering the highly efficient queue management and dataflow of hardwired hardware queues. To implement coherent hardware queues, push and pop memory primitives in software can be combined with configurable queue management units in hardware. Efficient synchronization and data movement can be achieved by performing queue management operations atomically at the location of the target queue, and by exploiting hints about locality and sharing patterns to optimize dataflow and queue placement. Perhaps most importantly, coherent hardware queues preserve the simplicity of a DRF consistency model because the push and pop operations can simply be thought of as new atomic RMW memory access types.

Efficient Coherent DMA: Direct memory access (DMA) is an important primitive for devices which request and operate on data in coarse-grained chunks, and it can be implemented in a coherent

or non-coherent manner. Coherent DMA offers simplicity and programmability benefits, but it can suffer from wasteful fine-grained coherence lookups if the target data chunk is together at a single location. We propose improving coherent DMA efficiency in two ways: reducing DMA network traffic by introducing coarse-grained request types, and reducing DMA lookup costs by introducing variable granularity state tracking. Although our current implementation of Spandex offers both word and line granularity request types, this flexibility can be extended to page granularity requests or larger. As long as the Spandex LLC is prepared to service such coarse-grained requests, this could significantly reduce the amount of request traffic required for DMA. Combined with owner prediction, page granularity requests could also be used to significantly reduce state lookups at the LLC. To reduce state lookups when owner prediction is not feasible, the LLC can track state at both page and word granularity. A page granularity coherence entry would only need to specify whether the target data can be found at one or multiple locations. If the target data can be found at one location (e.g. if it is owned in an HBM cache which tracks state at page granularity, or it is clean in memory), looking up each individual line of the page can be avoided.

Similar to other innovations described in this thesis, coherent hardware queues and efficient coherent DMA both identify common techniques which improve performance but add complexity in a specialized coherent memory system, and they offer similar benefits while preserving simplicity and programmability.

8.2.3 Future Impact

In the coming years, SoCs will increasingly rely on specialized accelerators and hardware-software co-design for delivering performance gains. Enabling the resulting diversity of specialized compute and memory storage elements to work together efficiently is therefore critical to the performance of future heterogeneous systems. The innovations described in this thesis present a framework for efficient heterogeneous integration through a coherent global address space supported by a simple DRF consistency model. We analyze emerging high-throughput accelerated workloads and show that efficient coherence is possible without resorting to complex changes to the consistency model. We formalize how atomic relaxation, which is so important to performance in many workloads, can be supported by future systems without sacrificing the simple SC nature of the consistency model. These coherence and consistency innovations represent a set of tools that can allow future

high-throughput workloads to achieve high cache efficiency while retaining a simple DRF consistency model. With Spandex, we present a simple and flexible interface for integrating these and other coherence strategies (including conventional CPU protocols) into a dynamically adaptable heterogeneous coherent memory system. We cannot predict what coherence strategies tomorrow's devices and workloads will prefer, so Spandex is designed to be flexible, composable, and extensible. As hardware-software co-design becomes more common and high level programming languages carry more information about communication patterns, Spandex will able to adapt to emerging innovations in hardware and software with new optimizations for important memory access patterns. Overall, we demonstrate that it is possible to offer specialization in the memory system to match specialization in emerging compute technology without incurring significant complexity for the hardware or software developer. Our focus on flexibility and simplicity allows this framework for heterogeneous coherence and consistency to remain relevant for years to come in a rapidly evolving hardware landscape.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Hazim Abdel-Shafi, Jonathan Hall, Sarita V Adve, and Vikram S Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 204–215. IEEE, 1997.

[3] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *TPDS*, pages 613–624, June 1993.

[4] Sarita Adve and Mark Hill. Weak Ordering – A New Definition. In *ISCA*, 1990.

[5] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, December 1993.

[6] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, pages 90–101, August 2010.

[7] N. Agarwal, T. Krishna, et al. GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator. In *ISPASS*, pages 33–42, 2009.

[8] Neha Agarwal, David Nellans, et al. Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence. In *HPCA*, pages 494–506, 2016.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, July 2014.

[10] Johnathan Alsop, Marc S Orr, et al. Lazy Release Consistency for GPUs. In *MICRO*, pages 1–14, 2016.

[11] Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *ISCA*, pages 172–182. IEEE, 2018.

[12] Johnathan Alsop, Matthew D Sinclair, Rakesh Komuravelli, and Sarita V Adve. GSI: A GPU stall inspector to characterize the sources of memory stalls for tightly coupled GPUs. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 172–182. IEEE, 2016.

[13] AMD Research. AMD's gem5 APU Simulator. http://www.gem5.org/wiki/images/7/7a/2015_ws_03_amd-apu-model.pdf.

[14] Craig Anderson and Anna R Karlin. Two adaptive hybrid cache coherency protocols. In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, pages 303–313. IEEE, 1996.

[15] A. Ariel, W.W.L. Fung, A.E. Turner, and T.M. Aamodt. Visualizing Complex Dynamics in Many-Core Accelerator Architectures. In *IEEE International Symposium on Performance Analysis of Systems Software*, pages 164–174, 2010.

[16] ARM. AMBA 5 CHI Architecture Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0050c/index.html, 2018.

[17] Ali Bakhoda, George L. Yuan, et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, pages 163–174, 2009.

[18] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 634–648, 2016.

[19] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The Problem of Programming Language Concurrency Semantics. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Berlin Heidelberg, 2015.

[20] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[21] Bradford M. Beckmann and Anthony Gutierrez. The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. In *MICRO Tutorial*, 2015.

[22] Jesse G Beu, Michael C Rosier, and Thomas M Conte. Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies. In *MICRO*, pages 226–236, 2011.

[23] Kristof Beyls and Erik H D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[24] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Multicast snooping: A new coherence method using a multicast address network. In *ISCA*, ISCA '99, pages 294–304, Washington, DC, USA, 1999. IEEE Computer Society.

[25] Hans-J. Boehm. Can Seqlocks Get Along with Programming Language Memory Models? In *MSPC*, 2012.

[26] Hans-J. Boehm. N3710: Specifying the absence of "out of thin air" results (LWG2265). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html, 2013.

[27] Hans-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm, 2013.

[28] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 68–78, New York, NY, USA, 2008. ACM.

[29] Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC, pages 7:1–7:6, 2014.

[30] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[31] Ian Bratt. The ARM® Mali-T880 Mobile GPU. In *IEEE Hot Chips 27 Symposium*, pages 1–27, 2015.

[32] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IISWC*, pages 141–151, 2012.

[33] C++. C++ Reference: Memory Order. `http://en.cppreference.com/w/cpp/atomic/memory_order`, 2015.

[34] Jason F Cantin, James E Smith, Mikko H Lipasti, Andreas Moshovos, and Babak Falsafi. Coarse-grain coherence tracking: Regionscout and region coherence arrays. *IEEE Micro*, 26(1):70–79, 2006.

[35] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to upc and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[36] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[37] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[38] Shuai Che, B.M. Beckmann, et al. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IISWC*, 2013.

[39] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[40] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th annual IEEE/ACM international symposium on microarchitecture*, pages 343–355. IEEE Computer Society, 2014.

[41] Liqun Cheng, John B Carter, and Donglai Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 328–339. IEEE, 2007.

[42] Byn Choi, R. Komuravelli, et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, pages 155–166, 2011.

[43] GenZ Consortium. Welcome to The Gen-Z Consortium! `http://genzconsortium.org`, 2017.

[44] MoChi Consortium. MoChi Architecture. `https://www.marvell.com/architecture/mochi/`, 2018.

[45] OpenCAPI Consortium. Welcome to OpenCAPI Consortium. `http://www.opencapi.org`, 2017.

[46] Alan L Cox and Robert J Fowler. Adaptive cache coherency for detecting migratory shared data. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 98–108. ACM, 1993.

[47] Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. The effects of granularity and adaptivity on private/shared classification for coherence. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(3):26, 2015.

[48] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, December 2011.

[49] David L Dill. The mur ϕ verification system. In *International Conference on Computer Aided Verification*, pages 390–393. Springer, 1996.

[50] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[51] HSA Foundation. HSA Platform System Architecture Specification. `http://www.hsafoundation.com/?ddownload=4944`, 2015.

[52] Benedict R. Gaster, Derek Hower, and Lee Howes. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *TACO*, 12(1):7:1–7:26, April 2015.

[53] Juan Gómez-Luna, Izzat El Hajj, et al. Chai: Collaborative Heterogeneous Applications for Integrated-Architectures. In *ISPASS*, pages 43–54, 2017.

[54] B.A. Hechtman, Shuai Che, et al. QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs. In *HPCA*, 2014.

[55] Derek R. Hower, Blake A. Hechtman, et al. Heterogeneous-Race-Free Memory Models. In *ASPLOS*, pages 427–440, 2014.

[56] CCIX Consortium Inc. Cache Coherent Interconnect for Accelerators (CCIX). `http://www.ccixconsortium.com`, 2018.

[57] D Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. D2ma: accelerating coarse-grained data transfer for gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 431–442. ACM, 2014.

[58] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. In *POPL*, 2017.

[59] Stefanos Kaxiras and James R Goodman. Improving cc-numa performance using instruction-based prediction. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 161–170. IEEE, 1999.

[60] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. *Lazy release consistency for software distributed shared memory*, volume 20. ACM, 1992.

[61] Ji Yun Kim and C. Batten. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. In *MICRO*, pages 75–87, 2014.

[62] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the Complexity of Hardware Cache Coherence and Some Implications. *ACM Transactions on Architecture and Code Optimization*, 11(4):37:1–37:22, December 2014.

[63] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Prakalp Srivastava, Maria Kotsifakou, Sarita V. Adve, and Vikram S. Adve. Stash: Have Your Scratchpad and Cache it Too. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 707–719, 2015.

[64] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access pattern-aware cache management for improving data utilization in gpus. *ACM SIGARCH Computer Architecture News*, 45(2):307–319, 2017.

[65] David A Koufaty, Xiangfeng Chen, David K Poulsen, and Josep Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1250–1264, 1996.

[66] Konstantinos Koukos, Alberto Ros, et al. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *TACO*, 13(1), 2016.

[67] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *ISCA*, 2015.

[68] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley 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*, pages 376–388. IEEE Computer Society, 2012.

[69] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 523–534. ACM, 2013.

[70] An-Chow Lai and Babak Falsafi. *Selective, accurate, and timely self-invalidation using last-touch prediction*, volume 28. ACM, 2000.

[71] Alvin R Lebeck and David A Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, pages 48–59, 1995.

[72] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 175–186, 2014.

[73] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[74] Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1677–1690, 2015.

[75] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646. IEEE Computer Society, 2014.

[76] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-OS interface. In *ASPLOS*, volume 50, pages 233–247. ACM, 2016.

[77] Daniel Lustig, Caroline Trippel, et al. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *ISCA*, pages 388–400, 2015.

[78] Peter S Magnusson, Magnus Christensson, et al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[79] Yatin A Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. Rtlcheck: verifying the memory consistency of rtl designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 463–476. ACM, 2017.

[80] Yatin A Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Ccicheck: using μhb graphs to verify the coherence-consistency interface. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 26–37. ACM, 2015.

[81] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, POPL, 2005.

[82] 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 Computer Architecture News*, 2005.

[83] Milo MK Martin, Pacia J Harper, Daniel J Sorin, Mark D Hill, and David A Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 206–217. ACM, 2003.

[84] Paul McKenney. Some Examples of Kernel-Hacker Informal Correctness Reasoning. In *Proceedings of the Dagstuhl Workshop on Compositional Verification Methods for Next-Generation Concurrency*, 2015.

[85] Shubhendu S Mukherjee and Mark D Hill. Using prediction to accelerate coherence protocols. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 179–190. IEEE Computer Society, 1998.

[86] Dan Negrut, Radu Serban, et al. Unified Memory in CUDA 6.0: A Brief Overview of Related Data Access and Transfer Issues. Technical report, University of Wisconsin-Madison, 2014.

[87] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[88] NVIDIA. CUDA SDK 3.1. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[89] Lena E Olson, Mark D Hill, and David A Wood. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *ASPLOS*, 2017.

[90] Molly A O'Neil and Martin Burtscher. Microarchitectural Performance Characterization of Irregular GPU kernels. In *IEEE International Symposium on Workload Characterization*, pages 130–139, 2014.

[91] Marc S Orr, Shuai Che, et al. Synchronization Using Remote-Scope Promotion. In *ASPLOS*, pages 73–86, 2015.

[92] Vassilis Papaefstathiou, Manolis GH Katevenis, Dimitrios S Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 325–334. ACM, 2013.

[93] Jean Pichon-Pharabod and Peter Sewell. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 622–633, 2016.

[94] Victor Podlozhnyuk. Histogram calculation in CUDA. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf`, 2007.

[95] Jason Power, Arkaprava Basu, et al. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *MICRO*, pages 457–467, 2013.

[96] Jeffrey B Rothman and Alan Jay Smith. Sector Cache Design and Performance. In *ISMASCTS*, pages 124–133, 2000.

[97] Nikolay Sakharnykh. Beyond GPU Memory Limits with Unified Memory on Pascal. `https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/`, 2016.

[98] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 73–82. IEEE Press, 2013.

[99] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *European Conference on Object-Oriented Programming*, pages 27–51. Springer, 2008.

[100] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *MICRO*, pages 647–659, 2015.

[101] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *ISCA*, pages 161–174, 2017.

[102] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs. In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, pages 239–249. IEEE, 2017.

[103] I. Singh, A. Shriraman, et al. Cache Coherence for GPU Architectures. In *HPCA*, 2013.

[104] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 37(3):69–80, 2009.

[105] Per Stenström, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. *ACM SIGARCH Computer Architecture News*, 21(2):109–118, 1993.

[106] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, Department of ECE and CS, University of Illinois at Urbana-Champaign, 2012.

[107] Jeff A. Stuart and John D. Owens. Efficient Synchronization Primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[108] Jeffrey Stuecheli, Bart Blaner, et al. CAPI: A Coherent Accelerator Processor Interface. *IBM JRD*, 59(1):7–1, 2015.

[109] Hyojin Sung and Sarita V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. In *ASPLOS*, pages 545–559, 2015.

[110] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *ASPLOS*, 2013.

[111] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *IEEE Micro's Top Picks from the Computer Architecture Conferences*, 2014.

[112] Herb Sutter. Atomic weapons: The C++ memory model and modern hardware. In *C++ and Beyond*, 2012.

[113] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*, pages 25–35. ACM, 2015.

[114] Josep Torrellas, HS Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *TOCS*, 43(6), 1994.

[115] Caroline Trippel, Yatin A Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. 52(4):119–133, 2017.

[116] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. ISCA, 2018.

[117] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[118] Zhenlin Wang, Kathryn S McKinley, Arnold L Rosenberg, and Charles C Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 199–208. IEEE, 2002.

[119] Hongbo Yang, Ramaswamy Govindarajan, Guang R Gao, and Ziang Hu. Improving power efficiency with compiler-assisted cache replacement. *Journal of Embedded Computing*, 1(4):487–499, 2005.

[120] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[121] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. Titanium: A high-performance Java dialect. *Concurrency and Computation: Practice and Experience*, 10(11-13):825–836, 1998.

[122] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[123] Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *ISCA*, volume 41, pages 547–558. ACM, 2013.