SPATIOTEMPORAL COHERENCE AND CONSISTENCY

by

Sooraj Puthoor

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

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2022

Date of final oral examination: 12/07/2022

The dissertation is approved by the following members of the Final Oral Committee:
Mikko Herman Lipasti, Professor, Electrical and Computer Engineering
Gurindar Singh Sohi, Professor, Computer Sciences
Mark Donald Hill, Professor, Computer Sciences
Joshua San Miguel, Assistant Professor, Electrical and Computer Engineering
Matthew David Sinclair, Assistant Professor, Computer Sciences

© Copyright by Sooraj Puthoor 2022
All Rights Reserved

To my Ammumma, Puthoor Karthyayani Amma, who took the first step towards ensuring a quality education for me.

Acknowledgements

First of all, I would like to express my gratitude towards my advisor Mikko Herman Lipasti for his guidance, support and patience throughout my time in graduate school. I firmly believe I could not have accomplished this much without his help. As my doctoral advisor and as a great human being, his influence has made me a better researcher and a better person. Equally important is the role of my parents, my Amma Sujatha Puthoor and my Achan Kulamburath Janardanan. Their love and support is my strength. And I thank them for inculcating values that I strive to live up to. I would also like to thank all the teachers who taught me over these years, all of you have helped me in completing this dissertation.

The feedback from my dissertation committee during the preliminary exam and the oral defense has greatly improved the presentation and the quality of this dissertation. I thank them for their feedback.

I also benefited from interactions with my colleagues at AMD Research, especially Brad Beckmann, Joseph Greathouse and Arkaprava Basu. I would also like to take this opportunity to thank John Alsop, Brad Beckmann, Alexandru Dutu, Anthony Gutierrez, John Kalamatianos, Onur Kayiran, Michael LeBeane, Matthew Poremba, Brandon Potter, Srikant Bharadwaj and many others who actively contributed to the development of the incredible simulator that I used in this work. I also want to express my appreciation to my management at AMD Research for their continued support during my PhD.

I enjoyed my interactions with Anirudh Mohan Kaushik who was the first to point out the applicability of this work to real-time systems. Discussions with Anirudh also helped me

reason about the deadlock avoidance rules in this dissertation. Special thanks go to Dibakar Gope for providing guidance to me during every step of the PhD process. I would also like to thank members of the PHARM research team Ravi Raju, Kyle Daruwalla, David Schlais, Gokul Ravi, Michael Mishkin, Rohit Shukla and Soroosh Khoram for their help and support.

Finally, thanks to my wife Janakey Devi Vayavinat Kizhakkepat Pathayapurayil for taking care of our family during my busy schedule completing this dissertation. My toddler girl Mahalakshmi Puthoor and my infant boys Anirudh Puthoor and Abhiram Puthoor also contributed to this dissertation by giving up their precious time with their father on few occasions, which I promise to give back upon its completion.

Table of Contents

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

List of Tables . . . . . **xiii**

Abstract . . . . . **xiv**

Chapter 1: Introduction **1**

1.1 GPU Protocol Requirements 3

1.1.1 Low Hardware Resource and Implementation Overhead 3

1.1.2 Low Coherence Message Overhead 4

1.1.3 Write Atomicity 4

1.1.4 Status of the State-of-the-art 5

1.2 Spatiotemporal Coherence 6

1.2.1 Low Hardware Overhead 7

1.2.2 Low Coherence Message Overhead 7

1.2.3 Write Atomicity 8

1.2.4 Generic Performance Optimizations 8

1.2.5 Entangled Coherence for Latency-Sensitive Scenarios 10

1.3 Lockstep Sequential Consistency 10

1.4 Dissertation Contributions 11

1.5 Dissertation Organization 12

Chapter 2: Background and Motivation 13

2.1 Heterogeneous-race-free Synchronization Operation 13

2.2 Limitations of Heterogeneous-race-free 15

2.2.1 Programmability Drawback 16

2.2.2 Portability Drawback 17

2.2.3 Performance Drawback 17

2.3 Need for Strong Model on GPUs 19

2.4 Related Work 20

2.4.1 GPU Coherence Protocols 20

2.4.2 Implementation of Strong Models on CPUs and GPUs 22

2.5 Chapter Summary 23

Chapter 3: Write Propagation by Address Bands and Epochs 24

3.1 Epoch Rules 25

3.2 Establishing Cache Coherence 26

3.2.1 Writes are Eventually Made Visible 27

3.2.2 Write Atomicity 27

3.2.3 Maintaining Program Order 27

3.3 Visibility of Writes 28

3.3.1 Epoch vs Modified State 28

3.3.2 Epoch vs Time Lease 29

3.3.3 Qualitative Comparison Against Traditional Protocols 30

3.4 Implications 31

3.4.1 Spatiotemporal Coherence and Write-through Caches 31

3.4.2 Providing Write Atomicity on GPUs 33

3.4.3 Correctness Invariant 33

3.4.4 Deadlocks 33

3.4.5 Stateless Coherency Controllers 34

3.4.6 Global Ordering with Spatiotemporal Coherence 34
3.4.7 Release Consistency Model on Spatiotemporal Coherence 35
3.5 Applicability of Spatiotemporal Coherence 35
3.6 Chapter Summary 37

Chapter 4: Performance Optimizations 38
4.1 Performance Optimizations 39
4.1.1 Epoch Skipping 39
4.1.2 Adaptive Bands 39
4.1.3 Multi-band Epochs 41
4.2 Entangled Coherence 42
4.2.1 Introduction to Entangled Coherence 42
4.2.2 Establishing Entanglement 43
4.2.3 Entangled Coherence vs. Spatiotemporal Coherence 45
4.2.4 Applicability of Entangled Coherence 46
4.3 Chapter Summary 47

Chapter 5: Consistency Model for Lockstep Execution 48
5.1 Program Order for Lockstep Execution 49
5.2 Lockstep SC 50
5.2.1 Lockstep SC model vs. *Extending SC to GPU* Models 52
5.2.2 Implicit Synchronization with Lockstep SC 54
5.2.3 Previous GPU SC Implementations 55
5.2.4 Interaction with the GPU Execution Model 55
5.3 Implications 56
5.3.1 Implications for Application Development 56
5.3.2 Implications for Low-level Software Development 56
5.3.3 Hardware Design Implications 57

5.4 Chapter Summary and Outline of the Second Part of this Dissertation 59

Chapter 6: Experimental Methodology 60

6.1 Simulator 60

6.2 Simulated System 61

6.2.1 GPU System 61

6.2.2 APU System 62

6.2.3 Heterogeneous SoC System 64

6.3 Workloads 65

6.3.1 Handwritten Workloads for Spatiotemporal Coherence 66

6.3.2 Handwritten Workloads for Lockstep SC 66

6.3.3 Software Pipeline Generator for Entangled Coherence 67

6.4 Chapter Summary 71

Chapter 7: Spatiotemporal Coherence for GPUs 72

7.1 Implementation 72

7.1.1 Release Consistency Implementation on Baseline GPU 73

7.1.2 Spatiotemporal Coherence Baseline 74

7.1.3 Epoch Skipping Implementation 79

7.1.4 Adaptive Band Implementation 80

7.1.5 Multi-band Epoch 82

7.1.6 Spatiotemporal Coherence and Cache States 83

7.1.7 Hardware Additions 84

7.2 Correctness of the Implementation 84

7.2.1 Ordering of Writes 86

7.2.2 Correctness with Multi-band Epoch Optimization 87

7.3 Evaluation 87

7.3.1 Execution Time 89

7.3.2 Sensitivity Analysis . . . 93
7.3.3 Bandwidth and Hardware Cost . . . 94
7.3.4 Scalability . . . 96
7.3.5 Comparison with Prior Work . . . 97
7.4 Chapter Summary . . . 98

Chapter 8: Entangled Coherence for Heterogeneous SoCs . . . 99
8.1 Implementation . . . 99
8.1.1 Band Formation . . . 100
8.1.2 Establishing Entanglement . . . 103
8.1.3 Operation under Entanglement . . . 105
8.1.4 Quasi-entanglement and Re-establishing Entanglement . . . 107
8.2 Correctness of the Implementation . . . 107
8.3 Evaluation . . . 110
8.3.1 Performance Evaluation . . . 113
8.3.2 Comparison against Spatiotemporal Coherence . . . 115
8.3.3 Bandwidth and Hardware Cost . . . 118
8.4 Chapter Summary . . . 119

Chapter 9: Lockstep SC for APUs . . . 120
9.1 Implementation Considerations . . . 120
9.1.1 Strong Ordering with Mutexes . . . 122
9.1.2 Extending Mutex Mechanism to OoO Cores . . . 123
9.1.3 Extending Mutex Mechanism to GPU . . . 123
9.2 Implementation . . . 124
9.2.1 Changes to GPU Memory System . . . 125
9.2.2 Changes to CPU Pipeline . . . 126
9.2.3 CPU Mutex Manager . . . 128

9.2.4 GPU Mutex Manager . . . 130
9.2.5 Mutex Pool . . . 131
9.3 Correctness of the Implementation . . . 132
9.3.1 Deadlocks and Mutex Acquisition . . . 133
9.4 Evaluation . . . 135
9.4.1 Application Performance Evaluation . . . 135
9.4.2 Increase in Bandwidth Consumption . . . 140
9.4.3 Sensitivity to Mutex Request FIFO size . . . 140
9.4.4 Hardware Cost . . . 141
9.5 Chapter Summary . . . 141

Chapter 10: Conclusions . . . 142
10.1 Summary and Conclusions . . . 142
10.2 Future Research Directions . . . 143
10.2.1 Extending Spatiotemporal Coherence to Realtime Systems . . . 144
10.2.2 Strong Models on Spatiotemporal Coherence . . . 144
10.2.3 Hierarchical Coherence Implementation . . . 146
10.2.4 Optimizations to Spatiotemporal and Entangled Coherence . . . 150
10.2.5 Speculative Execution . . . 151
10.3 Reflections . . . 152
10.3.1 Are We Over-reliant on CPU Ideas? . . . 152
10.3.2 What Should be the Simulation Methodology? . . . 153

Bibliography . . . 154

List of Figures

1.1 Spatiotemporal Coherence overview. 9

2.1 APU with tightly integrated CPU and GPU. 14
2.2 Command processor hooked up to the GPU cache hierarchy. 16
2.3 Impact of acquire operation. 18

3.1 STC epoch transition from epoch_A to epoch_B. 25

4.1 Adaptive band optimization demonstration. 40
4.2 Demonstration of entangled group formation. 43
4.3 Autonomous driving pipeline. 46

5.1 GPU program order. 49
5.2 *Extending SC to GPU* candidate models. 51
5.3 Implicit synchronization with Lockstep SC. 53
5.4 Coalescer adhering to SC. 58

6.1 Baseline APU system. 63
6.2 Pipeline generator use case: autonomous driving pipeline. 68
6.3 Pseudocode for an abstract pipeline stage. 69

7.1 Spatiotemporal Coherence baseline with epoch management unit. 73
7.2 Epoch and band implementation. 74

7.3 Epoch transition and synchronization. 77
7.4 Lazy invalidation hardware. 77
7.5 Adaptive band implementation. 79
7.6 Demonstration of correctness with a working example. 85
7.7 GPU execution time normalized to the baseline. 88
7.8 Decrease in the number of epoch transitions. 89
7.9 GPU L1 cache hit rate. 90
7.10 Sensitivity to blocked store queue size. 94
7.11 Bandwidth consumption of Spatiotemporal Coherence compared to the baseline. 95
7.12 Scalability of Spatiotemporal Coherence. 97

8.1 Band map. 100
8.2 Demonstration of band map compaction algorithm. 101
8.3 Entanglement map. 104
8.4 Operation after entanglement. 108
8.5 Performance of Entangled Coherence normalized to baseline. 112
8.6 Load hit rate comparison. 112
8.7 Uncacheable accesses with Entangled Coherence. 114
8.8 Cache hit rate comparison. 115
8.9 Performance comparison against Spatiotemporal Coherence. 116

9.1 Demonstration of atomic SC. 121
9.2 APU system with changes to support Lockstep SC. 124
9.3 Mutex manager and mutex pool. 127
9.4 CPU mutex manager operation. 129
9.5 Constraint graph demonstrating the correctness of Lockstep SC implementation. 132
9.6 Normalized application execution time. 134
9.7 Fraction of memory instructions committed by each benchmark. 134

9.8 Increase in zero commit cycles. 136
9.9 Mutex hit rate. 136
9.10 Normalized GPU execution time. 138
9.11 Hardware coherence overhead. 139
9.12 Bandwidth increase with Lockstep SC. 139

10.1 Interaction between epochs and mutexes. 145
10.2 Private cache with synchronization FIFO. 147
10.3 Efficiency of epoch relinquishing optimization. 150

List of Tables

3.1 Comparison of Spatiotemporal Coherence with other protocols 31

6.1 Simulation parameters 61
6.2 Changed GPU parameters 64
6.3 Workloads 65
6.4 APU benchmarks 67

7.1 State transition table 82
7.2 Spatiotemporal Coherence specific simulation parameters 88
7.3 Hardware cost of Spatiotemporal Coherence implementation 96

8.1 Hardware cost of Entangled Coherence implementation 118

9.1 Lockstep SC specific simulation parameters 135
9.2 Hardware cost of Lockstep SC implementation 141

Abstract

Strong memory consistency models such as sequential consistency (SC) impose ordering constraints that are intuitive and hence easy for programmers to reason about. However, the hardware and performance costs to support such models have been an impediment to adopt them. For example, the hardware coherence overhead to support strong models have limited their acceptability on GPUs. Consequently, GPUs today employ weaker models. Towards this, this dissertation first introduces a low overhead hardware coherence mechanism for GPUs called Spatiotemporal Coherence.

The proposed Spatiotemporal Coherence is a novel coherence implementation that implements coherence by establishing mutual agreement among the processors about modifiable addresses at any given time. Spatiotemporal Coherence satisfies all conditions of an implicit hardware-only coherence protocol with virtually no coherence overhead. We also propose several performance optimizations to Spatiotemporal Coherence including extending it to latency-sensitive scenarios with a variant called Entangled Coherence. Finally, realizing that a formally defined strong model does not exist for GPUs, we define and formalize a strong model on GPU called the Lockstep SC consistency model, a model very similar to SC but extended to support the lockstep execution paradigm of GPUs.

The most important contribution of this dissertation is the rule based Spatiotemporal Coherence protocol. Spatiotemporal Coherence's rules ensure that (a) all the processors are in agreement about the modifiable addresses at any given time, and (b) no potentially modifiable data is cached in the private caches of processors. Thus Spatiotemporal Coherence makes updates to an address visible to all the processors without coherence messages. To implement the first rule, that is to establish mutual agreement among processors, Spatiotemporal Coherence introduces the concept of epochs and address bands. An epoch represents a slice of time in which a group of addresses (called address bands or simply bands) associated with it can be modified. This assignment of write permissions to an epoch is fundamentally different from all previous coherence mechanisms assigning write permission to a processor core.

Chapter 1

Introduction

Graphics processing units (GPU) started as an accelerator targeting regular streaming applications without any data sharing. As such, they did not require any cache coherence support and caches were manged by the programmer. Consequently, irregular applications with memory access patterns that cannot be captured statically by programmers were unattractive on them. This changed with the arrival of hardware managed multi-level cache hierarchy in a GPU.

Along with these irregular applications came the need for coherent data sharing and the initial implementations provided coherence by disabling the GPU's private caches. Naturally, this solution came with performance penalty for applications requiring caching support. Thus arose the need for cache coherence protocols in a GPU.

A cache coherence protocol can be a directory-based protocol or a snooping-based protocol [99]. However, directory based coherence protocols are preferred by systems with large number of processors owing to their scalability advantages [75, 80, 99, 106]. The directory-based protocols invalidate private copies of an address before granting write-permission of that address to the modifying processor. To filter unnecessary invalidation messages, the directories also keep a sharer list that tracks the private copies of each address. With the sharer list optimization, the invalidation messages can now be targeted only towards this

subset of private caches.

Singh *et al.* has shown that these traditional invalidation based directory coherence protocols will incur significant performance penalty in a GPU with invalidation, eviction, and recall messages overwhelming its memory subsystem [98]. Current GPUs avoid strict hardware coherence implementations because of this coherence traffic overhead [80, 87, 98]. Instead, GPUs today implement caches that maintain coherence at release consistency (RC) synchronization markers inserted by the software [27, 37, 54, 62] and adhere to the heterogeneous-race-free (HRF) models. Here, we refer to such cache coherence implementations as software-managed cache coherence.

While the cache design becomes simple and coherence traffic is eliminated with software-managed cache coherence, programmers will have to deal with the non-intuitive nature of weak models, complicating the programming of such systems and increasing the likelihood of software bugs. There is a plethora of work showing weak models are notoriously difficult to program and are extremely challenging to verify [6, 7, 21, 28–30, 53, 95, 109, 110]. The reordering permitted by the weak models results in a non-intuitive sequence of memory operations, often resulting in programming bugs that are hard to detect. Additionally, there has been a series of prior work attempting to formalize the weak models of the POWER and ARM architectures, only to get it wrong in their initial attempts [8, 20, 36, 72, 82, 92]. Alglave *et al.* observed that some implementations of weak models violate programming guides and vendor documentation guarantees [5].

The HRF memory model [54] requires a programmer to (a) specify the scope of communication and (b) explicitly initiate the communication. That means HRF goes one step further than RC, requiring the programmer for explicitly managing the communication scope whereas RC only demands the programmer to identify explicit synchronization operations. While it is true that most of these complexities can be abstracted away with synchronization routines, the low level software developer who writes these synchronization routines will still have to deal with these intricacies. A strong model like sequential consistency (SC) neither

asks the programmer for visibility scopes nor demands the programmer to insert explicit synchronization primitives. Despite the clear programmability advantages with strong models, the major hindrance for providing a strong model on a GPU is the non-availability of a low-overhead hardware coherence mechanism with write-atomicity [2] support.

In this dissertation, we ask the question: can a strong model be implemented on a GPU with low overhead? Towards this, we propose a novel Spatiotemporal Coherence protocol that retains all the advantages of a software-managed cache coherence while also satisfying the properties of a strict hardware cache coherence protocol [2]. Specifically, this dissertation solves the low-overhead coherence implementation with write-atomicity conundrum. A strong model implementation itself is left as future work. While answering this question, we also make the observation that a well-defined and formal strong model does not exist for lockstep executing threads in a GPU. So, we define and formalize Lockstep SC, a consistency model with strict ordering requirements for lockstep executing threads.

In this chapter, we first list the desired GPU coherence protocol requirements to support strong models (Section 1.1). Then we provide an overview of the Spatiotemporal Coherence and discuss how it satisfies these requirements (Section 1.2). In Section 1.3, we give a brief overview of the Lockstep SC consistency model We list the contributions of this dissertation in Section 1.4 and conclude this chapter with the dissertation organization in Section 1.5.

1.1 GPU Protocol Requirements

1.1.1 Low Hardware Resource and Implementation Overhead

GPUs gain performance benefits by instantiating massive amount of compute resources enabling massive levels of parallelism. So, while adding any new GPU feature, a GPU architect always assesses the trade-off of adding a new hardware structure for that feature against adding one more compute unit. Thus chip real estate is an expensive resource in a GPU and a hardware resource heavy coherence implementation consuming chip real estate for logic or

for storage is not acceptable in a GPU.

1.1.2 Low Coherence Message Overhead

An AMD graphics core next (GCN) GPU can have up to 40 wavefronts running concurrently in each compute unit (CU) with each wavefront having 64 threads [12]. Thus each CU can have up to 2560 active threads and a GPU coherence protocol should thus employ minimum coherence messages to implement coherence without which the number of coherence messages will explode leading to bandwidth bottleneck and consequent performance degradation.

In a traditional directory based protocol, the invalidation, eviction and recall traffic contributes to the coherence message overhead. A traditional protocol typically employs a probe filter to track the sharers of a cache block and to filter out coherence invalidation messages. The recall and eviction messages are required for maintaining the sharer list in the probe filter. While adding a probe filter reduces coherence traffic, it increases the hardware resources needed for maintaining coherence and goes against the low hardware resource requirement discussed in section 1.1.1. Additionally, given the streaming nature of GPU workloads, the evictions are quite frequent in a GPU and the eviction notifications soon dominate the network traffic [80, 98]. Traditional coherence protocols were designed for multi-core CPUs and with 2560 threads per CU, the coherence traffic originating from even a single CU is much higher than a multi-core CPU. Hence the need to think about new ways of implementing coherence for GPUs.

1.1.3 Write Atomicity

Write atomicity - the property that the writes to the same address should be visible to all processors in the same order - is a precondition for implementing strong models [2]. Thus a GPU coherence protocol that can enable a strong model must provide write atomicity. The traditional protocols provide write-atomicity. However, they provide it by invalidating the sharers of a cache block and granting exclusive modification rights to a processor. Like

discussed in section 1.1.2, invalidation messages increase the coherence messages in a system and go against the low coherence message overhead requirement.

1.1.4 Status of the State-of-the-art

In this section, we will analyse some state-of-the-art coherence proposals from prior works to determine where they stand in terms of these requirements.

The invalidation approaches including the traditional directory based protocols are not scalable because of the need for invalidation messages for every write and hence cannot be employed in a GPU [49, 98]. The timestamp-based coherence protocols [87, 98, 100] provide write atomicity and avoid invalidation messages. These protocols avoid invalidation messages by getting a time lease on an address and self-invalidating that address after the time lease has expired. However, they have a fundamental requirement to store lease information in every cache block for their correct operation. This increases their storage overhead. Also, this storage overhead scales with the number of cores. Although their scalability is better than the invalidation based protocols, the metadata storage overhead to store lease information limits its applicability to a GPU.

Denovo protocols [35, 94] avoid invalidation messages and provide write-atomicity. However, it is a hybrid software-driven hardware coherence protocol that identifies read-only regions but with the help of software assistance and/or programmer inserted hints. Such models either put the onus on the programmer to annotate the access type of a data region and/or add constraints to the model such as data-race-freedom for their correct operation. It also has the additional overhead of tracking coherence states, albeit less states than a traditional protocol, in the caches thus increasing the hardware storage overhead.

Software coherence [49, 54]) is scalable with virtually no hardware overhead but requires software assistance for its correct operation. It does not provide write atomicity and hence not capable of supporting strong models. Spatiotemporal Coherence introduced in the next section has comparable hardware requirements as the software coherence but provides write

atomicity without any software assistance.

1.2 Spatiotemporal Coherence

The key insight of the Spatiotemporal Coherence is that a coherence protocol can be implemented with virtually no overhead by establishing mutual agreement between processors about the modifiable addresses within a given time window. To establish mutual agreement, Spatiotemporal Coherence introduces the concept of epochs and address bands. An epoch represents a slice of time in which a group of addresses (called address bands or simply bands) associated with it can be modified. Thus, if all processors agree on the current epoch, write propagation can then be achieved with a set of simple caching rules. This assignment of write permissions to an epoch is fundamentally different from all previous coherence mechanisms assigning write permission to a processor core. An epoch has temporal properties and an address band has spatial properties, hence the name Spatiotemporal Coherence.

Spatiotemporal Coherence works with these four simple epoch rules (ER):

1. ER1: At any (logical) time, all processors should be in the same epoch.
2. ER2: Addresses in band_A can be written only in epoch_A.
3. ER3: Addresses in band_A cannot be cached in epoch_A.
4. ER4: Addresses in a band can be read in any epoch.

The ER1 ensures all processors are synchronized regarding the current epoch. ER2 along with ER1 establishes mutual agreement on the addresses modified in a time window, that is an epoch. ER3 is the caching rule which states that potentially modifiable addresses must not be cached. This means during the window of modification, a store will write its data to the shared cache and a load will always read the most up-to-date data from the shared cache thus ensuring correct write propagation.

1.2.1 Low Hardware Overhead

Spatiotemporal Coherence is an attempt to mimic the GPU software coherence but without software assistance. Software coherence makes the caches coherent only at RC synchronization boundaries. However, without software assistance, a hardware coherence mechanism has no means to know when to synchronize accesses. Hence it has to assume that every access must be synchronized across all the CUs. Towards this, an epoch represents an acquire-release synchronization point but for a subset of addresses (band) associated with it. As such, the only requirement for providing coherence is to map all addresses to bands and rotate granting epochs to those bands. Thus, Spatiotemporal Coherence can be viewed as a self-synchronizing software coherence mechanism. As such, there are no significant hardware overheads. The only additional overhead compared to the software coherence mechanism is the synchronization overhead, that is the overhead to rotate and synchronize epochs.

1.2.2 Low Coherence Message Overhead

The synchronization involves sending the new epoch to CUs and waiting for their acknowledgement confirming the epoch changes. However, frequent epoch synchronizations may amplify this overhead. Towards this, we leverage the latency-tolerant nature of GPU and synchronizes epochs only at some fixed interval of time. The writes may be delayed with such an approach but the fixed time interval can be chosen such that it does not impact the performance of GPUs. This approach makes epoch synchronization less frequent thus adding almost negligible message overhead over the software coherence mechanism as we will show in the evaluation of this protocol (Section 7.3). These epoch synchronization messages replace all coherence traffic such as invalidation, eviction and recall traffic of the traditional coherence protocols.

1.2.3 Write Atomicity

The caching rules of the Spatiotemporal Coherence ensure that a store will only modify the data in a shared cache and a load will always read the most up-to-date from the shared cache. Consequently, all the writes to the same address will be visible to all CUs thus providing write atomicity. Additionally, to maintain write atomicity, Spatiotemporal Coherence stalls conflicting reads to pending stores waiting for their epoch to retire. That is forwarding of a pending store is not permitted.

1.2.4 Generic Performance Optimizations

Skipping epochs with no writes enables epochs with writes to retire their writes faster. This also lead to less number of epoch synchronizations. Grouping bands with read-only addresses can amplify the impact of epoch skipping by avoiding granting epochs to these read-only bands. Finally, allowing multiple-bands to retire in an epoch can also facilitate faster write completion. We discuss these three optimizations - epoch skipping, adaptive band and multi-band epoch optimizations- in detail in Chapter 4.

Figure 1.1 provides an overview of Spatiotemporal Coherence with the epoch skipping and adaptive band optimizations. Figure 1.1(a) shows a three epoch system with the caching rules for each epoch. Since writes to bands are only allowed in their epoch, the epochs are rotated in round-robin fashion so that every band gets an opportunity to write. Figure 1.1(b) shows a simple scheme to map address bands to epochs. In this example, all addresses with the same contiguous 2-bits from bit position 8 are grouped to the same band. This scheme can be extended to N contiguous bits starting from any bit position of an address for the band formation. It should be noted that the address to band mapping can be implemented in multiple ways with several other hashing schemes to map an address to a band including software/programmer defined address to band mapping. But we are forming bands using the simple hardware-only contiguous N-bit address to band mapping scheme shown in that figure. Figures 1.1(c) and (d) shows two optimizations that improves the performance of

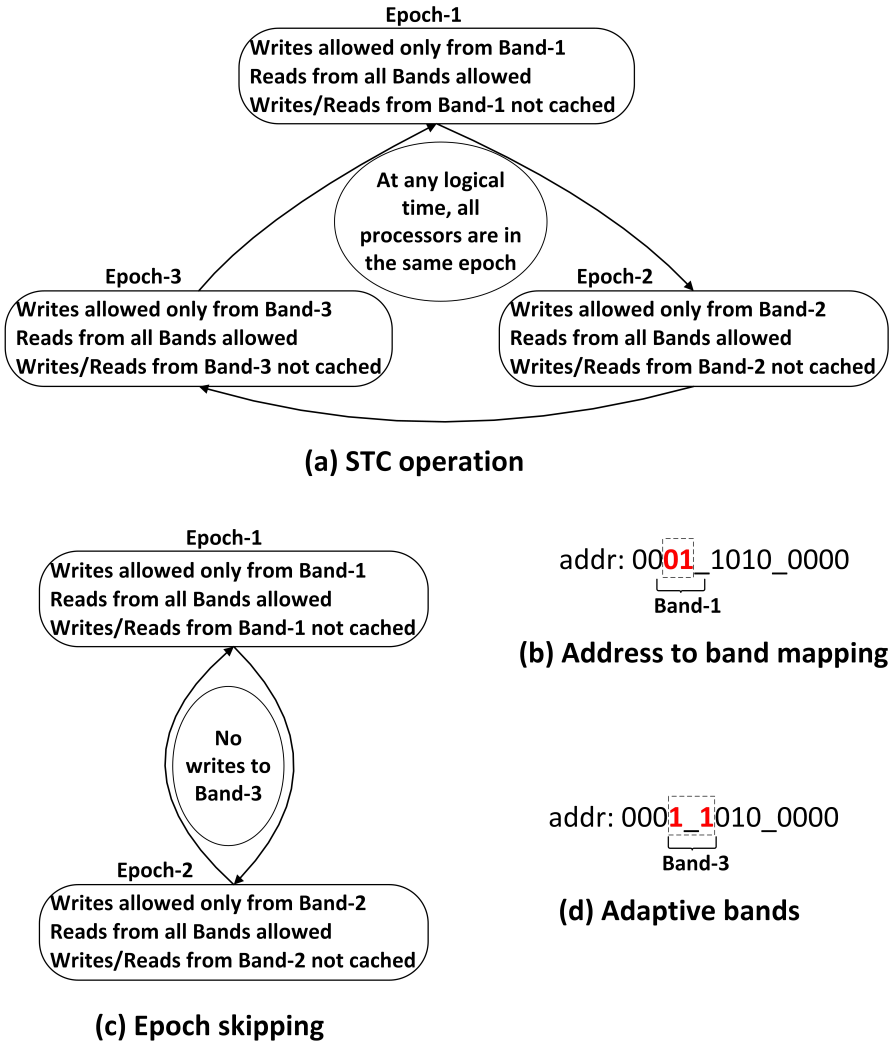


Figure 1.1: Spatiotemporal Coherence overview. (a) demonstrates Spatiotemporal Coherence's epoch rules with three epochs. A detailed explanation of these epoch rules can be found in Section 3.1. (b) shows as simple strategy for mapping addresses to band. In that example, 2 contiguous address bits starting from bit position 8 are used for grouping addresses into bands. (c) demonstrates the epoch skipping optimization. The Epoch-3 is not granted because there are no writes to its band Band-3. The adaptive band optimization is shown in (d). The address to band mapping is changed dynamically to group read-only addresses together thus preventing granting unnecessary epochs to these bands. Chapter 4 discusses these optimizations in detail.

STC. Specifically, Figure 1.1(c) skips epochs with no writes so that the epochs with writes can retire their writes faster. Figure 1.1(d) shows changing the epoch to address band mapping dynamically so that read-only bands can be separated from the rest to amplify the impact of the epoch skipping optimization.

1.2.5 Entangled Coherence for Latency-Sensitive Scenarios

Although Spatiotemporal Coherence is a GPU cache coherence protocol, it can be extended to certain latency-sensitive scenarios. We explore this possibility with an extension of Spatiotemporal Coherence called Entangled Coherence. Entangled Coherence groups processors that share data into an entangled group. This enables localized epoch synchronization among a subset of processors in that entangled group bringing down the synchronization overhead. Also, knowledge about the sharing pattern of data enables private accesses to be grouped into a private epoch and then excluded from coherency overheads. Naturally, such an optimization only works if the application has a fixed sharing pattern or a long application phase with a fixed sharing pattern such as an autonomous driving pipeline or mobile SoC pipelines [52, 68]. More details about Entangled Coherence can be found in Section 4.2.

1.3 Lockstep Sequential Consistency

SC ordering rules are defined for individual threads with independent program order [99]. But GPUs execute threads in lockstep and it is unclear whether the program order should be defined for individual GPU threads or if the lockstep executing threads share the same program order. Towards this, we first argue that all the lockstep executing threads share a common program order. Since SC is defined for threads with independent program order, we make the observation that SC ordering rules do not provide ordering for memory operations originating from a lockstep execution. Towards this, we formally define Lockstep SC memory model that defines the ordering rules for memory operations from lockstep execution.

Lockstep SC allows lockstep memory operations to be unordered in memory. More details about Lockstep SC can be found in Chapter 5.

1.4 Dissertation Contributions

The following are the most important contributions of this dissertation:

- **Proposes a coherence protocol that works on the principle of mutual agreement between processors:** This dissertation proposes Spatiotemporal Coherence that provide cache coherence by establishing mutual agreement between processors about the modifiable addresses within a given time window called an epoch. This protocol also shifts processor-centric write permission to an epoch-centric write permission significantly reducing the hardware resource overhead, implementation complexity and coherence message overhead while providing write atomicity.

- **Proposes a well-defined formal strong model for lockstep executing threads:** We introduce Lockstep SC, a consistency model that is as strong as SC but formally defined to accommodate the lockstep execution model. Prior to this model, we are not aware of any strong consistency model that defined the ordering rules for lockstep execution.

- **Implements and evaluates the coherence protocol along with its optimizations:** We provide an implementation and a detailed evaluation of Spatiotemporal Coherence along with its optimizations on a GPU. We also provide an implementation and evaluation of Entangled Coherence - the extension of Spatiotemporal Coherence for latency-sensitive scenarios - on an emulated heterogeneous SoC.

- **Provides an implementation and evaluation of the newly defined strong model on an APU SoC:** We provide an implementation and evaluation of Lockstep

SC across an entire APU SoC. In the process, we provide a very detailed and well-performing implementation of a strong model on a dynamically scheduled out-of-order CPU core.

1.5 Dissertation Organization

This dissertation is divided into two parts. The first part describes the concepts of Spatiotemporal Coherence and Lockstep SC consistency model. It begins with this introductory chapter (Chapter 1) and moves to the chapter that gives a background of the heterogeneous-race-free model, discusses its limitations and motivates the need for strong models on GPUs (Chapter 2). That chapter also provides a survey of the related work. Chapter 3 introduces the concept of Spatiotemporal Coherence and discusses its GPU coherence design implications. Chapter 4 presents various performance optimizations of Spatiotemporal Coherence and introduces Entangled Coherence as a performance optimization to latency-sensitive scenarios. Lockstep SC concepts are introduced in Chapter 5 and that concludes the first part of this dissertation.

The second part of the dissertation focuses on the implementation and evaluation of the concepts introduced in the first part. We start with the experimental methodology chapter (Chapter 6) that provides details about the simulator, the simulated system and the workloads used for evaluation. Chapters 7, 8 and 9 implements and evaluates Spatiotemporal Coherence, Entangled Coherence and Lockstep SC respectively. These chapters are similarly organized with each chapter providing the implementation details followed by a proof of the implementation and finally a detailed evaluation of the implementation. We provide concluding remarks and reflect on our work in the last chapter (Chapter 10) along with providing some directions for future work.

Chapter 2

Background and Motivation

Heterogeneous-race-free (HRF) is a well defined memory model that enables localized synchronization in an accelerated processing unit (APU) system-on-chip (SoC) by providing scoped synchronization operations [54]. The scopes define the level to which synchronization operations should be made visible. The Heterogeneous System Architecture (HSA) foundation [40] and many commercial APUs have adopted this model [27, 62]. In this chapter, we first discuss the details of HRF synchronization operation on an APU system (Section 2.1). Then we discuss the limitations of this model (Section 2.2) and motivates the need for a stronger model on GPUs (Section 2.3). We survey the related research on this topic in Section 2.4 before concluding this chapter with a chapter summary (Section 2.5).

2.1 Heterogeneous-race-free Synchronization Operation

HRF requires explicit synchronization instructions such as load-acquire (ldAcq) and store-release (stRel) to communicate between threads [43, 49, 54]. A ldAcq is a downward memory fence that ensures memory operations in program order are visible only after that acquire operation is made visible whereas stRel guarantees that all memory operations from that thread are visible before that release operation. A formal definition of these synchronization operations can be found elsewhere [43, 54, 99].

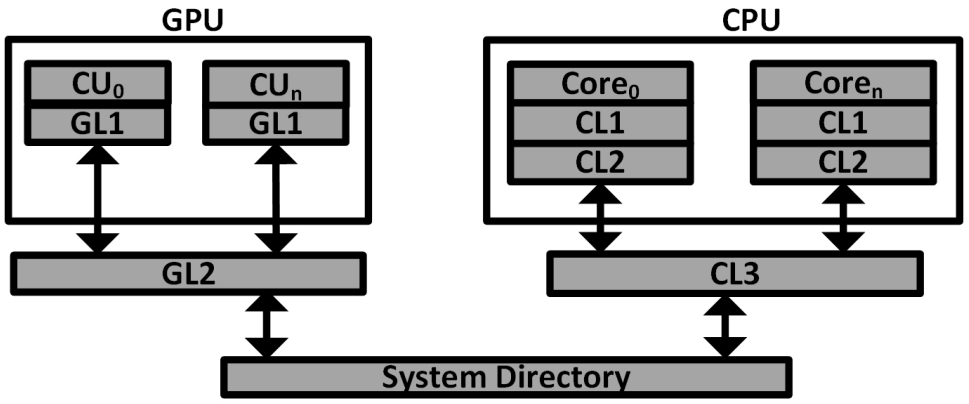


Figure 2.1: APU with tightly integrated CPU and GPU. The CPU and GPU are coherently connected by a system directory. The scoped synchronization operation of the HRF defines the levels to which synchronization must be made visible. The workgroup scoped operations are made visible only to the private L1 caches (GL1) of GPUs whereas a device scoped synchronization makes it visible to the shared GPU L2 (GL2). Only a system scoped synchronization will make the updates from CPU and GPU visible to each other.

Figure 2.1 shows an APU with tightly integrated CPU and GPU. Each compute unit (CU) of the GPU has a private L1 cache (GL1) and a shared L2 cache (GL2). GPU caches enforce HRF by invalidating the valid cachelines on a ldAcq operation and write back all outstanding writes on a stRel operation [11, 54]. For a typical APU, only the GL1s are managed by HRF synchronization operations and GL2s are kept coherent by invalidation messages from system directory [27, 47, 62]. Additionally, write-through GL1s do not need to write back data during a stRel operation. However, even with write-through GPU caches, a stRel operation must wait for all write-through completion acknowledgements (cmplAcks) from the memory hierarchy. For a work-group or device scoped synchronization, these acks are generated from the shared GL2 whereas a system scoped synchronization must have the acks from the system directory. While a stRel waits for all cmplAcks, the GPU pipeline need not wait for the cmplAck of individual stores because HRF enforces memory fences only at synchronization operations.

A ldAcq operation invalidates all valid cachelines in the GL1 cache, which can be achieved

by resetting their valid bits. A subsequent load request to the GL1 after a ldAcq will find the cache in invalid state and will forward that load to the GL2 for the most recent copy of the data. Thus, both device scoped and system scoped ldAcq invalidates the GL1 whereas a work-group scoped does not because GL1 is already the common visibility point for the threads in a work-group (all wavefronts in a workgroup are typically scheduled on the same CU).

Since GL1s are made coherent with ldAcq operation, the GL2 do not forward invalidation messages from the directory to these GL1s. GL2s perform non-silent evictions to keep it inclusive with probe filter but evictions from GL1s are silent.

Discrete GPUs

The CPU and GPU in a discrete GPU setup typically operates on disjoint address spaces and hence the synchronizations discussed above are only applicable to the work-group and device scopes. When the data needs to be transferred from a CPU to a GPU buffer or vice-versa, a system scope acquire or release operation is performed respectively on the GPU. The acquire operation invalidates the GL1 and GL2, and the release operation writes the dirty lines in the GL2 to the GPU memory thus ensuring coherent data transfer of that buffer.

2.2 Limitations of Heterogeneous-race-free

Weak models are challenging to program and verify [6, 7, 21, 28–30, 53, 95, 109, 110]. They rely on programmers to explicitly insert synchronization primitives to initiate communication. However, there are testimony from programmers about their struggles with relaxed model's synchronization primitives, including spending days trying to get code with just two addresses and four accesses to work [95].

The HRF model not only asks the programmer to initiate the communication but also specify the scope of communication [54]. Today, the scopes are defined in terms of GPU

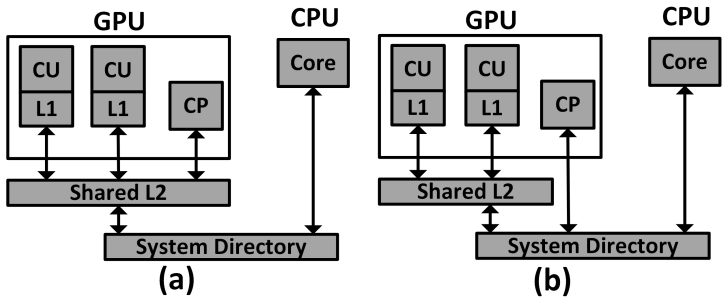


Figure 2.2: Command processor hooked up to the GPU cache hierarchy. There is a trend towards using the command processor for tasks other than launching work on a GPU [64, 79]. Since scopes are defined in terms of GPU execution hierarchy, even though command processor resides in the GPU, the scope to synchronize GPU threads and command processor threads is unclear.

execution hierarchies. For example, work-items within the same work-group synchronize through work-group scope, and work-items from different work-groups synchronize through device scope (scopes are present in other models as well [70]). While a GPU programmer already has the functional correctness burden with the weak model, these scopes now add the performance burden also on the programmer.

The *3p* criteria - programmability, performance, and portability - was introduced by Adve [4] to evaluate memory models. Next we will demonstrate that HRF pales in all three criteria.

2.2.1 Programmability Drawback

While use of scopes are well-defined for synchronizing between work-items of a GPU, the synchronization between work-items and threads running on other processing elements on the same GPU is not clearly defined. For example, the Graphics Core Next (GCN) GPUs have a command processor (CP) to launch work on the GPU [13] and since CP threads are not part of the GPU execution hierarchy, the synchronization scope between them and work-items are not well-defined by HRF.

Figure 2.2(a) shows the CP and CUs sharing the same shared L2 in a GCN GPU [13]. Recent studies have suggested using the CP for GPU initiated networking [64] and for enabling fine-grained task scheduling [79]. Enabling both these capabilities require CP threads and work-items to synchronize frequently. In the absence of a clearly defined scope for such synchronization in the HRF, in practice, programmers choose the level of the memory hierarchy where the synchronization should occur as the scope. In this case, the shared L2 is the scope of the synchronization, which is the same as device scoped synchronization. This scenario thus exposes HRF's drawback of forcing a programmer to think about the underlying cache hierarchy design while writing synchronizing programs, an observation made by Sinclair *et al.* in their work [94] as well.

2.2.2 Portability Drawback

While the scenario mentioned in previous section is a programmability drawback of HRF, the implications go far beyond that. Figure 2.2(b) shows an alternative implementation of hooking the CP into the cache hierarchy of an APU. In this scenario, the CP is hooked to the system directory. Hence, synchronization between CP threads and work-items through device scope is no longer possible. This scenario exposes a portability issue with HRF. While HRF can still guarantee synchronization with system scope in this scenario, the synchronization code has to be manually modified based on this new cache hierarchy.

2.2.3 Performance Drawback

The HRF invalidates the private caches before a kernel launch (acquire operation) to ensure stale data produced in previous GPU phases is not read by the new kernel. However, for workloads with read-only data reuse across kernel launches, this kernel launch induced invalidations results in performance loss. Figure 2.3 compares the private L1 cache hit rate of a few workloads with acquire operation against a version without acquire operation. The details of these workloads can be found in the methodology section (Section 6.3.1). Suppressing the

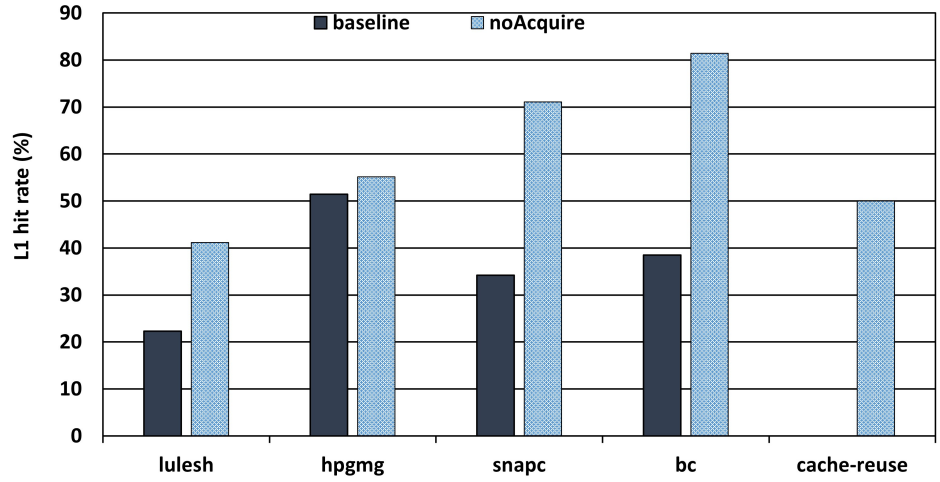


Figure 2.3: Impact of acquire operation. Benchmarks with read data reuse across kernel invocations can potentially benefit from selective invalidations that target only modified data at kernel boundaries .

acquire operation nearly doubled the hit rate for most of these benchmarks. As we will show in the evaluation of Spatiotemporal Coherence (Section 7.3.1), these benchmarks exhibited read data reuse across kernel launches and HRF's invalidation operation at the beginning of a kernel launch invalidated these reused data from their caches resulting in performance loss.

Suppressing the Acquire Operation

The HRF managed private caches cache both modified (by a peer core) and unmodified data. An acquire operation is needed only to invalidate the modified data and the baseline can get away by suppressing the acquire operation if there is a way to ensure that the private cache is only caching unmodified data. But distinguishing between the modified and unmodified data is difficult. Hence the HRF resorts to the entire cache invalidation.

2.3 Need for Strong Model on GPUs

Like discussed in the previous section (Section 2.2), HRF has programmability, portability and performance drawbacks. These drawbacks are more pronounced while supporting future use cases such as GPU initiated networking [64] and enabling fine-grained task scheduling [79]. Moreover, in heterogeneous systems with fine-grain communication between CPUs and GPUs, having the same consistency model for both CPU and GPU makes it easy for a programmer to reason about complex work sharing interactions between CPU and GPU phases of a program. This is especially true with APU/GPU programming moving towards single source languages like C++ AMP [74] or HC [16]. HRF does not provide any advantage in this regard as well. Prior work has attempted to translate software written for one memory model to target heterogeneous processors adhering to different memory models [71]. However this came with significant performance overhead (up to 77% [71]) even when translating between compatible memory models.

Since strong models such as SC present a highly intuitive model for programmers to reason about producer-consumer interactions, it can significantly reduce the software complexity of writing heterogeneous applications. Strong models also achieve low overhead fine-grain synchronization without any release consistency synchronization overheads like invalidation and write-back of caches [43, 49, 54]. Their relatively simple ordering guarantees reduces the hardware design and verification complexity of these systems as well. Because of these reasons, we argue that GPUs must support strong models.

In this dissertation, we focus on removing the major hindrance for providing a strong model on a GPU - that is the non-availability of a low-overhead hardware coherence mechanism with write-atomicity support. Spatiotemporal Coherence proposed in this dissertation retains all the hardware advantages of a software-managed cache coherence implementation while also satisfying the properties of a strict hardware cache coherence protocol. Spatiotemporal Coherence and its implications are discussed in detail in Chapter 3.

We also observed that a formal strong model did not exist for GPUs. Towards this,

we define and formalize a strong memory model called Lockstep SC in this dissertation. Lockstep SC is similar to SC but extends ordering rules to lockstep executing threads. A detailed discussion about Lockstep SC can be found in Chapter 4.

In the next section, we survey some relevant related works.

2.4 Related Work

2.4.1 GPU Coherence Protocols

The reduction of coherence traffic in GPUs has been the subject of many studies in the past [11, 22, 49, 80, 87, 98, 100]. Prior work has also explored coherence across multiple GPUs [88, 105]. Power *et al.* and Basu *et al.* proposed tracking addresses at larger granularities called regions to reduce the number of invalidation messages [22, 80]. Both these techniques track coherence permissions over a larger region granularity. Spatiotemporal Coherence's bands are different from regions in three ways: (a) regions track coherence permissions whereas bands do not, (b) regions are modified by processors with exclusive/private permissions but STC bands are modified exclusively in its epoch (processor versus epoch centric write permissions), and (c) regions are contiguous addresses whereas grouping addresses into a band is entirely implementation-specific. More details about this can be found in Section 3.3.

There are mechanisms like Denovo protocol [35] that identify read-only regions but with the help of software assistance and/or programmer inserted hints. Such models either put the onus on the programmer to annotate the access type of a data region and/or add constraints to the model such as data-race-freedom for their correct operation. Spatiotemporal Coherence does this without any software assistance and without imposing any constraints on the model.

Sinclair *et al.* provided a taxonomy of commonly used CPU/GPU coherence protocols based on (a) invalidation initiator, and (b) tracking up-to-date copy of the data [95]. Since

invalidation initiator is an epoch in Spatiotemporal Coherence and the up-to-date copy is maintained by write-through operation, Spatiotemporal Coherence differentiates itself from those protocols.

Alisafaee proposed Spatiotemporal Coherence tracking directory [9] that tracks private data at region granularity. However, this proposal is an efficient directory design to reduce the directory capacity, not a coherence protocol.

The timestamp-based coherence protocols [87, 98, 100, 107] eliminate invalidation messages by self-invalidation based on expiring time lease. Although this method eliminates invalidation traffic, it suffers from scalability and storage overheads. We evaluated Spatiotemporal Coherence against timestamp-based coherence. The results of our evaluation can be found in Section 7.3.5. The concept of programmer specified timestamps is also employed by Jeffrey *et al.* [57] to detect conflicts and order task commits in their implementation of a scalable architecture for ordered parallelism.

The Racer protocol by Ros and Kaxiras provides store atomicity without invalidations and per-processor write permission [89]. Racer achieves this by dynamically detecting read-after-write data races and self invalidating the racing reader's cache on a data race. Spatiotemporal Coherence avoids the need for dynamic race detection and self invalidation by restricting writes to an epoch and making the potentially modifiable data non-cacheable in private caches.

Spatiotemporal Coherence simplifies the coherence mechanism by establishing a mutual agreement among all the processors about the modification of addresses. Ros and Kaxiras also leveraged a similar concept for non-speculative store coalescing without deadlocks [90]. The globally agreed lexicographical order for acquiring cache permissions prevents cyclic cache permission dependencies making the implementation deadlock-free. Ibrahim *et al.* [55] proposed a shared L1 architecture for GPUs that obviates the need for coherence among the L1-sharer CUs. However, when such a system scales with shared L1 clusters, the Spatiotemporal Coherence can provide coherence across these clusters with negligible hardware

cost.

2.4.2 Implementation of Strong Models on CPUs and GPUs

Numerous researchers have investigated efficient support for SC such that it matches the performance of weaker models on a CPU [3, 24, 31, 32, 44–46, 66, 67, 86, 97, 103]. These approaches can be classified as speculative approaches and non-speculative approaches. The speculative approaches require expensive hardware mechanisms to speculate beyond consistency model imposed boundary and to rollback when mis-speculated. The non-speculative approaches [3, 46, 67, 97] enforce SC by ordering only conflicting accesses and significantly reduce the hardware overhead. These non-speculative approaches differ in the way they detect conflicts and order conflicting accesses. For example, Singh *et al.* [97] rely on compiler techniques to identify conflicting accesses whereas Adve *et al.* [3] rely on underlying cache coherence mechanism to detect and serialize conflicting accesses. Conflict ordering proposed by Lin *et al.* [67] uses an augmented write buffer to keep track of active memory operations (memory operations retired from the ROB, but whose predecessors are pending in the global memory order) and uses tagged coherence messages to detect and order conflicting active memory operations. The atomic SC proposed by Gope and Lipasti [46] relies on mutexes to detect and serialize conflicting accesses. Hence, it does not require expensive associative look ups or compiler support, and does not interfere with the underlying coherence implementation. Our implementation of Lockstep SC also relies on mutexes for ordering conflicts.

Academic researchers have proposed implementing SC on GPUs in the past [50, 87, 96]. While all these implementations produced promising results, they differed in their underlying assumptions. For example, Singh *et al.* [96] implemented SC on a baseline that has a MESI cache coherence protocol while Ren and Lis [87] and Singh *et al.* [98] has demonstrated that a CPU like coherence *read-for-ownership* protocol is not suitable for GPUs. To this end, we use a realistic baseline GPU cache coherence and evaluate our implementation against

this realistic baseline. We also show the bandwidth overhead of maintaining coherence using invalidation messages in GPU caches.

While most past proposals implemented SC on either the CPU or GPU, there are only a few prior works that implemented strong models on a heterogeneous SoC like an APU. Alsop *et al.* [10] proposed Spandex that implemented SC-for-DRF [1] on a heterogeneous system. However, unlike our Lockstep SC implementation, SC-for-DRF requires a programmer to insert synchronization primitives similar to HRF.

2.5 Chapter Summary

In this chapter, we described the operation of heterogeneous-race-free model (Section 2.1), demonstrated its limitations (Section 2.2) thus motivating the need for a strong model on GPUs and a low overhead coherence mechanism to support it (Section 2.3). Then we provided a literature survey of the related research in this area (Section 2.4). The next three chapters introduce the Spatiotemporal Coherence (Chapter 3), discuss some performance optimizations and introduce the Entangled Coherence (Chapter 4), and define and formalize Lockstep SC consistency model (Chapter 5).

Chapter 3

Write Propagation by Address Bands and Epochs

In this chapter, we first define the epoch rules of the Spatiotemporal Coherence (STC) that are necessary for correct write propagation (Section 3.1). Then we discuss how STC's write propagation is different from other protocols (Section 3.3) and how the epoch rules satisfy the requirements of cache coherence (Section 3.2). We elaborate on the implications of STC protocol in Section 3.4. Finally, we list the application characteristics that are better suited for STC and also the characteristics that are less suited for STC in Section 3.5 before concluding this chapter with a chapter summary (Section 3.6).

STC achieves write propagation with address bands and epochs. An address band is simply a group of addresses. All addresses in an address band, hereafter referred simply as a band, share the same write coherence permission. An epoch is defined as a time window in which a band is given write permission. Each band will have at least one epoch associated with it. For example, band_A will have at least an epoch_A when stores to addresses belonging to band_A are allowed to be issued to the memory hierarchy.

An epoch differs from the modified (M) state of a coherence protocol although both grant write permission [99]. An epoch grants write permission *to the band associated with it* but M

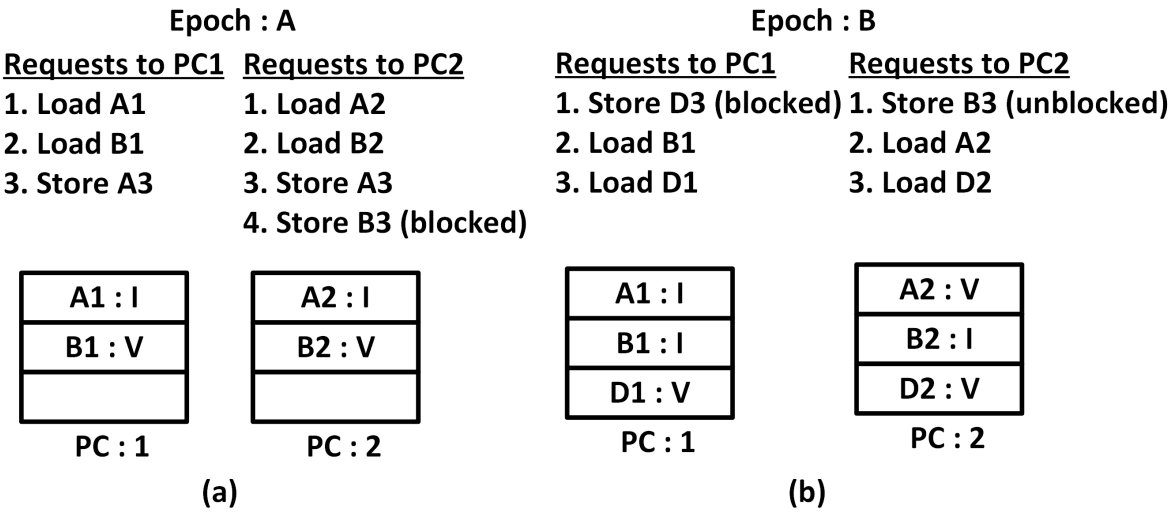


Figure 3.1: STC epoch transition from epoch$_A$ to epoch$_B$. The loads A1 and A2 belonging to band$_A$ are not cached in the private caches in that band's epoch epoch$_A$. Store B3 is blocked because it does not belong to that epoch. Later, when the epoch transitioned to epoch$_B$, the loads B1 and B2 of band$_B$ are invalidated from the private caches, and the blocked store B3 is unblocked and issued to the memory.

state grants write permission *of an address to a processor*. STC shifts the processor-centric write permissions in traditional protocols to epoch-centric write permissions. Additionally, to preserve the single-writer-multiple-reader (SWMR) invariant, an M state is exclusive with only the processor in M state having a copy of the cache block. An epoch does not provide exclusivity guarantees either.

3.1 Epoch Rules

To ensure write propagation, STC works with four simple epoch rules (ER):

1. ER1: At any (logical) time, all processors should be in the same epoch (or epochs).

2. ER2: Addresses in band$_A$ can be written only in epoch$_A$.

3. ER3: Addresses in band$_A$ cannot be cached in epoch$_A$.

4. ER4: Addresses in a band can be read in any epoch.

ER1 makes sure all processors observe the same epoch (or epochs) and there is no conflict among the processors about the current epoch (or epochs). ER2 ensures that writes are not cached in private caches and are visible to the memory side cache or memory. ER3 ensures that a potentially modifiable cache block is not cached in a private cache and hence no processor will see stale data as long as all processors are observing the same epoch. Private caches can still cache data from bands when the processors are not in that band's epoch. ER4 ensures that reads are never blocked.

We will illustrate these rules with the help of an example. Figure 3.1 shows the epoch transition from epoch_A to epoch_B for two private caches PC1 and PC2. That figure also shows the loads and stores arriving at these caches in that epoch. Figure 3.1(a) shows both PC1 and PC2 after encountering the memory request sequence in epoch_A. The loads to band_A (A1, A2) are issued to memory but they are not cached adhering to ER3. Thus, during epoch_A, both C1 and C2 does not cache any addresses of band_A. However, loads to band_B (B1, B2) are cached in the private caches and the addresses B1 and B2 are in valid (V) state. The store to address A3 belonging to band_A is issued to the memory but the store to band_B is blocked by ER2. Figure 3.1(b) shows the epoch transition to epoch_B. The band_B addresses (B1,B2) are invalidated and the store to B3 from epoch_A is now unblocked and issued to the memory. The loads to epochs other than epoch_B can now cache data in private caches.

We discuss how STC satisfies the conditions of cache coherence in the next section.

3.2 Establishing Cache Coherence

The two conditions associated with cache coherence protocols are: (a) a write is eventually made visible to all processors and (b) the writes to the same address should be visible to all processors in the same order (write atomicity) [2]. STC's epoch rules are sufficient to satisfy

these two conditions.

3.2.1 Writes are Eventually Made Visible

ER1 ensures that all processors agree on the current epoch. Since only the band associated with it can be modified in that epoch, there is mutual agreement among the processors about the potentially modifiable addresses. This is the key to STC establishing cache coherence. Once this mutual agreement is established, the two other epoch rules ER2 and ER3 ensures that no potentially modifiable data is cached in the private caches. Consequently, every store updates the memory side cache (or memory) and every load belonging to the current epoch reads the most up-to-date data from that shared cache. Thus STC satisfies the first condition of making a write eventually visible to all processors.

3.2.2 Write Atomicity

Write atomicity comes naturally with STC. The very essence of the STC rules is that no modifiable data is cached in the private caches. Thus, every single write to a band and consequently every address in a band is visible to all processors in the system. Additionally, to maintain write atomicity, STC stalls conflicting reads to pending stores waiting for their epoch to retire. That is forwarding of a pending store is not permitted.

3.2.3 Maintaining Program Order

In addition to the above-mentioned two conditions, a third condition is often associated with cache coherence protocols: a coherence protocol should inform the writing processor about the completion of a store[2, 98]. Satisfying this condition is required to maintain program order between the write instruction and younger instructions. Towards this, STC sends an acknowledgment back to the writing processor when a store is completed. However, unlike traditional protocols that have to wait for the invalidation of private copies before sending

this acknowledgment back, STC can immediately respond with the acknowledgment when the store reaches the memory side cache (or memory). Maintaining program order between the write instructions is a requirement for strong models such as sequential consistency (SC) and total store order (TSO), and not a requirement for a weak model such as release consistency[2]. Thus, even though STC sends the write completion ack, a strong model implementation will use it for enforcing ordering but a relaxed model implementation does not need to do it.

3.3 Visibility of Writes

Traditional coherence protocols make an update to an address visible to the processors by either (a) invalidating the copy in the private caches by sending invalidation probes (invalidation-based protocols) or (b) updating the value in all private caches by update messages (update-based protocols). STC uses neither of these mechanisms, and hence cannot be characterized under these traditional classifications. Next we compare the epoch-based write propagation against the M state of a traditional protocol and also against the time lease mechanism of a timestamp-based protocols.

3.3.1 Epoch vs Modified State

Traditional protocols adhere to the single-writer-multiple-reader (SWMR) invariant and this invariant mandates an exclusive modifier processor (single-writer) for correct write propagation. An address in the Modified state (M state) in a processor's cache means that processor has exclusive write permission to that address[99]. STC changes this processor centric write permission to an epoch centric write permission. The direct implication of such an approach is that the modification privileges are now associated with an epoch and multiple processors can participate in the modification of a band in its epoch, provided there is a proper policy in place to satisfy the conditions of cache coherence. The STC epoch rules discussed

earlier establishes these policies. With these epoch rules, all processors observe the same epoch and all processors agree to the band that is modified in each epoch. This mutual agreement among processors enables each core to identify the band (set of addresses) that are potentially modifiable at any given time without notifications–like invalidation or update messages–from the coherence protocol. Thus, just by knowing the epoch and synchronizing the epochs across all the processors, STC completely eliminates the need for explicit messages to propagate a write.

3.3.2 Epoch vs Time Lease

STC differs substantially from related prior work in timestamp-based coherence protocols [87, 93, 98, 100, 106]. The timestamp-based coherence protocols avoid invalidation messages by getting a time lease on an address and self-invalidating them after the time lease has expired. These protocols still follow the idea of an exclusive modifier or single writer for an address. The logical timestamp versions of these protocols also enforce the single-writer property, although in logical time. Thus, these timestamp-based protocols also grant processor-centric write permissions. STC does not grant write permission to a processor but to an epoch.

Moreover, storing time lease information as metadata in every cache block consumes space in both the private and shared caches [87, 98, 100, 106]. STC does not require cache blocks to store their lease information or any other metadata in their caches or controllers. Thus STC is stateless whereas timestamp-based protocols are not.

Additionally, since these protocols need to store the timestamp metadata of all cache blocks in a shared cache, the shared cache is forced to be designed as an inclusive cache [98]. Since the size of an inclusive cache increases with the number of private caches, an inclusive cache is not scalable. Also, inclusive caches face the challenge of cache evictions. Since the timestamp based protocols do not employ recall messages to solve the shared inclusive cache eviction challenge, they resort to the use of MSHRs to store the lease information of the evicted blocks [98]. This again leads to scalability challenges because the MSHR demand

will increase, perhaps dramatically. An alternate solution is to use a non-inclusive cache and then keep a single lease information for all the untracked cache blocks [87, 100]. However, this solution is not scalable either because as the number of private caches increases, the fraction of tracked cache blocks decreases in the non-inclusive cache. Since all the untracked cache blocks now share the longest lease among them, a modifying processor will be forced to advance its current logical time beyond this long lease time resulting in lease expiry induced invalidations of cache blocks in its private cache.

3.3.3 Qualitative Comparison Against Traditional Protocols

Table 3.1 compares STC with some of the GPU protocols proposed in the past [49, 87, 98, 100]. The GPUIn is an invalidation-based GPU protocol that propagates a new write by invalidation messages [49, 98]. This is not a protocol optimized for GPUs [49, 98] but added to this comparison to highlight the characteristics of an invalidation-based protocol. RCC is the relativistic coherence proposed by Ren and Lis [87], GTSC is the coherence protocol for GPUs proposed by Tabbakh *et al.* [100] and TCS is the coherence protocol proposed by Singh *et al.* [98]. All these protocols build on top of the timestamp-based coherence with the former two using logical timestamps proposed by Yu and Devadas [106]. The SRC is our baseline protocol with RC-managed caches and used in many previous studies [11, 47, 49, 84].

The GPUIn approach is not scalable because of the need for invalidation messages for every write and hence cannot be employed in a GPU. The timestamp-based coherence protocols provide write atomicity and avoid invalidation messages. Hence scalability of these protocols is better than the GPUIn protocol. However, the need for a large inclusive cache to store lease information or the increasing number of untracked blocks with a non-inclusive cache limits their scalability. They also have the additional storage overhead of tracking the lease information of every cache line in the metadata of a cache. SRC is scalable but requires software assistance and cannot provide write atomicity. STC is scalable and provides write

Table 3.1: Comparison of Spatiotemporal Coherence with other protocols

	Software assisted	Need exclusive write permission	Metadata overhead	Scalability	Write atomicity
GPUIn [49, 98] Coherence by invalidation messages	No	Yes	No	No (invalidation messages)	Yes
TCS [98] Temoporal coherence	No	Yes	Yes (timestamp)	No/Limited (inclusive cache)	Yes (in the strong version)
RCC [87] Relativistic coherence	No	Yes (in logical time)	Yes (timestamp)	Limited (untracked blocks)	Yes
GTSC [100] Global timestamp coherence	No	Yes (in logical time)	Yes (timestamp)	Limited (untracked blocks)	Yes (in the strong version)
SRC [11, 47, 49, 84] Release consistency managed caches	Yes	No	No	Yes	No
STC **Spatiotemporal coherence**	**No**	**No**	**No**	**Yes**	**Yes**

atomicity without software assistance and without incurring metadata overhead.

3.4 Implications

STC has significant coherence design and performance implications. This section discusses its implications on systems with write-through caches (Section 3.4.1) and how it can enable write atomicity to GPUs (Section 3.4.2). We then discuss the correctness invariant and provide the deadlock avoidance rules in Sections 3.4.3 and 3.4.4. Finally, we conclude this section by going over some of its characteristics such as stateless coherency controllers (Section 3.4.5), global ordering (Section 3.4.6), and invalidation-free acquire operation with RC model (Section 3.4.7).

3.4.1 Spatiotemporal Coherence and Write-through Caches

In a traditional protocol with write-through caches, a write request can be issued directly to the memory without that processor acquiring the write permission. However, the write can complete only after the sharers are invalidated by invalidation messages. STC does not require these invalidation messages and the memory-side cache (or memory) can send the write completion ack immediately after seeing the write request. Thus, STC is not a mere optimization of the traditional protocol for write-through caches.

STC and GPU Caches

STC with write-through private cache has profound implications. Typically, GPU caches are write-through with a simple VI (valid/invalid) protocol for coherence [11, 12, 49, 80, 98]. Previous studies have suggested that a write-through L1 cache performs much better than a write-back L1 for GPUs because of the streaming data access pattern of a GPU [63, 98]. Even commercial GPUs employ write-through private caches [12, 27, 62, 77].

However, implementing strict hardware coherence on a GPU is challenging. A Graphics Core Next (GCN) GPU has 40 wavefronts (warps) with 64 workitems single compute unit (CU) and a typical APU with 8 CUs will have 20,480 threads issuing concurrent requests to the cache hierarchy [12, 27, 62]. Although coalescing techniques employed by GPUs reduce the number of requests issued to the memory [84], the number of requests is still overwhelming for a hardware coherence mechanism to handle [80, 98]. For example, the directory must send an invalidation probe to all sharer CUs (each CU hosts a private L1) on receiving a write request and these sharers must respond back to this request with an acknowledgment. To precisely keep track of the sharers, the private L1s must send eviction notification to the directory. Given the streaming nature of GPU workloads, the evictions are quite frequent and the eviction notifications along with the invalidation traffic will soon dominate the network traffic [80, 98]. The directory size will also explode if inclusively tracking all the sharers, while a non-inclusive directory will introduce significant recall traffic [98]. Because of the above mentioned difficulties with hardware coherence mechanism, GPUs today adhere to release consistency model (RC), which makes the L1 caches coherent only at synchronization points with the help of software inserted synchronization primitives [11, 12, 49, 80].

STC eliminates all the above-mentioned challenges of hardware cache coherence in a GPU. STC does not rely on invalidation messages for making the caches coherent. Since invalidations are not needed, sharers need not be tracked and since the need for sharer tracking is eliminated, the private L1 caches can now perform silent evictions. Thus, STC implements hardware coherence without invalidation traffic, eviction notifications, and recall

traffic. Thus STC is a natural fit for GPUs.

3.4.2 Providing Write Atomicity on GPUs

In traditional protocols, write atomicity is implemented with invalidation or update messages that propagate a new write to all processors in the system. This approach is not easy to scale and can incur a severe performance penalty in a many-core system like a GPU [80, 98]. However, STC does not have this limitation. Since the epoch rules ensure that the processors have agreed on the band to be modified in each epoch and since the epoch themselves are synchronized, the processors do not cache a potentially modifiable address in the cache. Thus, the processors need not be notified about a write but they fetch the updated value from a memory-side cache (or memory). Thus, the STC implementation provides write atomicity without any additional coherence traffic.

3.4.3 Correctness Invariant

SWMR invariant is needed for correct write propagation in traditional coherence protocols. However, verifying against this invariant for correctness is necessary only if the writes are performed in private caches. STC does not allow writes to be performed in private caches. STC is a rule-based system and adhering to the rules provides correctness. Hence, the correctness invariant is to check against the STC rules, which is straightforward.

3.4.4 Deadlocks

Epoch rule ER2 states that a band can write only in its epoch. Thus, a write reaching the cache controller on a different epoch is blocked till its epoch arrives. Consequently, a processor may deadlock if an epoch for a write never arrives. Thus, to prevent deadlocks and to ensure forward progress for all writes, an STC implementation should adhere to the following deadlock avoidance rules (DARs):

1. DAR1: There must be at least one epoch for every band.

2. DAR2: All epochs must eventually terminate.

3. DAR3: All epochs must be granted repeatedly.

DAR1 ensures that all bands have at least one epoch, DAR2 ensures that an epoch will eventually end and DAR3 ensures that every band's epoch will be eventually granted. Thus, they ensure a write will eventually complete.

3.4.5 Stateless Coherency Controllers

STC allows processors to take localized decisions based on the current epoch and based on the information about that epoch's band. However, for STC to function correctly, all processors should be in the same epoch. Irrespective of the mechanism employed to synchronize the current epoch across all processors, this synchronizing mechanism can be stateless. Since STC does not rely on invalidation messages for write propagation, there is no need for a centralized structure to track the sharers of an address. Additionally, since processors do not own write permissions in STC, tracking global cacheline permissions is not necessary either. Contrast this to a traditional coherence directory that stores the sharer list and global cacheline states for maintaining coherence or a snooping protocol that relies on broadcast snoops and responses for maintaining coherence [99]. Thus STC eliminates the need for broadcasts or structures such as a global directory. In directory based coherence protocols, even a null directory (stateless directory) has to broadcast invalidations on receiving a write request, whereas STC does not need invalidation messages for maintaining coherence.

3.4.6 Global Ordering with Spatiotemporal Coherence

Global ordering is dictated by a directory in traditional protocols. The directory serves as the ordering point for coherence requests. However, since coherence messages are completely eliminated in STC, an explicit ordering point like a directory or snooping bus is not necessary.

An ordering point is needed when the processors do not have any agreement on the modifying addresses and the time of modification. With epochs and bands, the processors have mutually agreed upon the time and addresses they are modifying. As such, the only requirement is to synchronize epochs across all processors.

3.4.7 Release Consistency Model on Spatiotemporal Coherence

In an RC GPU, every kernel launch invokes an acquire and a release operation at the beginning and end of the kernel respectively. These RC-managed caches rely on private cache invalidation for an acquire operation and store completion acks for a release operation as discussed in section 2.1. However, if an RC model is built on top of STC, the cache invalidation during an acquire operation can be completely avoided. The STC's caching rules ensure no stale data is ever resident on a private cache. Thus STC enables an invalidation-free acquire operation for GPUs enabling an invalidation-free kernel launch. A more detailed discussion about this can be found in the implementation section 7.1.1.

3.5 Applicability of Spatiotemporal Coherence

STC allows writes only in their corresponding epochs and hence will delay write completion. While this is acceptable for latency tolerant accelerators such as GPUs, this may not be acceptable for latency sensitive processors such as CPUs. Hence, STC is better suited for GPUs. Among GPU applications, STC is generally suited for applications with sparse writes (write overhead from STC will not be pronounced). and application phases with high read-only data locality (taking advantage of STC's invalidation-free acquire operation). Based on these observations, we list few GPU applications that are better suited for STC and few others that are less suited for STC.

Applications Better Suited for STC

Applications with back-to-back kernel launches and with read-data reuse across these kernels can benefit from STC. The baseline GPU invalidates the private caches at the beginning of every kernel launch (acquire operation), but the STC caches provide invalidation-free acquire operations thus allowing caching of data across kernel launches and consequently increasing the private cache hit rate. This scenario is observed in snapc application and is discussed in the result section 7.3.1. Also, task-parallel applications are a class of emerging GPU applications from the high-performance computing (HPC) domain that launches GPU kernels as dependent tasks and these applications, because of their high data reuse across kernels, can get benefited from STC [23, 26, 83, 85, 104]. These task-parallel applications launch several concurrent kernels. With the baseline RC model, private cache invalidation from each kernel launch will adversely impact caching efficiency of all concurrently running kernels as shown by Puthoor *et al.* [85]. However, STC's invalidation-free acquire operations will benefit these applications as well. Additionally, some future GPU applications like in-memory databases [61] can also perform well with STC. These applications spend a significant amount of time in the indexing phase doing hash table lookups. Hence, these applications have high read locality with sparse writes [61] and thus better suited for STC. Non-volatile memory (NVM) applications are typically written to hide long-latency memory writes [76] and hence the write latency addition from STC will not add significant overhead. But STC can provide better caching efficiency for read datasets of these applications as well.

Applications Less Suited for STC

One of the limitations of STC as compared to the baseline RC-managed cache coherence is its inability to avoid coherence overhead for private data. In a GPU, compiler produces spills to private segment when the data set cannot fit into the available registers. Spills may also be generated by a compiler for increasing the GPU occupancy by reducing per-workgroup register pressure. Irrespective of the reason for spills, STC may not be well-suited

for applications with large amount of spills (private data). However, STC can leverage some of the software-assisted classification from prior work [22, 48, 96, 97] and allow the private data writes to complete without waiting for their epoch. This optimization is left for future work. However, an extension of STC called entangled coherence discussed in the next chapter's section 4.2 leverages this private data classification but without software assistance to evade STC's epoch rules .

3.6 Chapter Summary

We introduced the novel Spatiotemporal Coherence in this chapter. We discussed its epoch rules in Section 3.1, how these epoch rules satisfy cache coherence in Section 3.2 and how STC's write propagation is different from other protocols in Section 3.3. The implication section (Section 3.4) details how STC is a natural fit for GPU cache coherence, its correctness invariant, how it enforces deadlock freedom and how it simplifies cache controller design. Finally, we list the application characteristics that are better suited for STC and also the characteristics that are less suited for STC in Section 3.5.

In the next chapter (Chapter 4) we will discuss some performance optimizations for STC and extend it to latency-sensitive scenarios with Entangled Coherence variant.

Chapter 4

Performance Optimizations

This chapter presents the performance optimizations for Spatiotemporal coherence (STC). The generic performance optimizations are discussed in Section 4.1. Section 4.2 introduces the latency-sensitive variant of STC called Entangled Coherence. Finally, Section 4.3 concludes this chapter with a chapter summary.

STC can be viewed as a protocol that allows a band to cache data only in the absence of any coherent update to it. Thus, if a processor enters an epoch in which no writes are issued to the corresponding band, the band unnecessarily lost its opportunity to cache data, adversely impacting the performance of the processor. Another potential performance pitfall with STC is a band with read-only addresses. Since epochs are only needed to grant write permissions to a band, a read-only band does not need an epoch. Hence, granting an epoch for a read-only band is unnecessary and since that band is non-cacheable in that epoch, it is undesirable as well from a performance standpoint. To solve these performance pitfalls, we first introduce three optimizations in this chapter: (a) epoch skipping, (b) adaptive bands, and (c) multi-band epochs. We also introduce an extension of spatiotemporal coherence for write latency sensitive scenarios called entangled coherence in this chapter.

4.1 Performance Optimizations

4.1.1 Epoch Skipping

The epoch skipping optimization grants an epoch only if there is a demand for it. A demand to an epoch indicates that at least one write is pending to that epoch from a processor. With epoch skipping, the epochs that did not have any requests to them are skipped. Thus epoch skipping avoids the performance penalty of granting epochs to a band with no demand. Since a read-only band will not issue any write-request, epoch skipping will avoid granting permissions to read-only epochs as well.

Epoch skipping does not violate deadlock avoidance rules DAR2 and DAR3 (Section 3.4.4). These rules state that an epoch must eventually terminate and all epochs must be eventually granted. Epoch skipping can be treated as an optimization that terminates the current epoch adhering to DAR2 and grants zero cycles for all epochs with no write demands, adhering to DAR3. Thus, epoch skipping only prioritize the epochs that need to be granted immediately from the epochs that are needed only in the future.

The implementation details of epoch skipping can be found in Section 7.1.3. As we will show in that section, this optimization is relatively straightforward and does not require any major hardware changes either. However, two new epoch messages are needed - one message to request the epoch and another to acknowledge that request - for its implementation but they barely increase the bandwidth consumption (Section 7.3.3).

4.1.2 Adaptive Bands

The effectiveness of epoch skipping is amplified if the bands are clearly separated into read-only bands and non-read-only bands (write and read-write bands). The adaptive band optimization dynamically changes the addresses in a band to regroup read-only addresses together. Coupled with the epoch-skipping implementation, this dynamic identification of read-only datasets enables sustained caching of blocks across otherwise unnecessary write

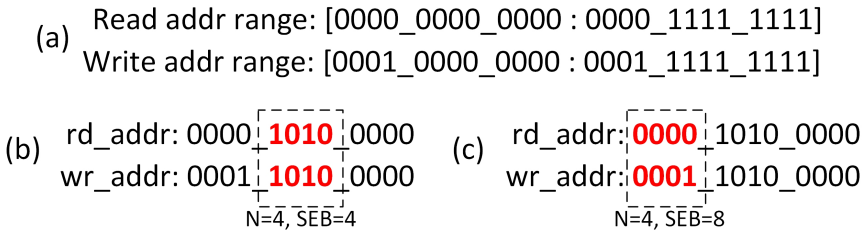


Figure 4.1: Adaptive band optimization demonstration. (a) shows the disjoint read and write address ranges. (b) shows two addresses of different access types mapped to the same band when 4 contiguous address bits (N=4) starting from address bit position 4 (Start Epoch Bit, SEB=4) are used for address to band mapping. (c) shows how adaptive band optimization can change the address to band mapping by changing the SEB to 8 thereby grouping read-only addresses to their own band.

epochs, since there are no pending writes to such read-only bands.

Figure 4.1 demonstrates the adaptive band optimization. The read and write address ranges are shown in Figure 4.1(a). Suppose we are grouping addresses with the same value for the 4 bits starting from address bit 4 (N=4 and start epoch bit SEB=4 respectively in Figure 4.1(b)) into the same band. With such a static mapping, the read address 0x0**A**0 and write address 0x1**A**0 shares the the same band 0xA as shown in Figure 4.1(b). Now, when an epoch is granted to band 0xA for performing a write to address 0x1A0, all the addresses in that band including the read-only address 0x0A0 will be unnecessarily invalidated from the private caches. However, if the the start bit for address to band mapping is moved to 8 (SEB=8) as shown in Figure 4.1(c), both read and write address ranges get a different band (0x0 and 0x1 respectively). Since the previously discussed epoch skipping optimization never grants an epoch to read-only bands, these read-only addresses are now safe from epoch invalidations. Thus, the adaptive band optimization dynamically changes the address to band mapping to filter out the read-only bands from epoch invalidation.

While Figure 4.1 demonstrated one possible way to identify the read-only bands, the design space for implementing this optimization is huge. For example, there are prior works that rely on software identification of read-only regions [35, 97]. Since our objective with

STC is to come up with a hardware-only coherence implementation, we did not explore these software solutions. However, prior works can still be used for implementing the adaptive band optimization, and they may even perform better than the approach presented here because they can reduce or even eliminate the time taken for learning the correct address to band mapping. The implementation details of the adaptive band optimization presented here can be found in Section 7.1.4.

4.1.3 Multi-band Epochs

Once adaptive band optimization filters out the read-only bands, epochs are granted only to write or read-write bands. Multi-band optimization grants epochs corresponding to multiple bands at the same time so that stores belonging to these bands can be issued without getting blocked. Thus, with this optimization, stores are completed faster.

Increasing the band range by combining write-only or read-write bands together is an alternate approach that may achieve the same benefits as multi-band optimization. However, increasing the band range may also result in combining multiple dissimilar bands together, adversely impacting performance. For example, suppose there are three adjacent independent bands A (0000_**1010**), B (0000_**1011**) and C (0000_**1100**). Suppose B is a read-only band whereas A and C are either write-only or read-write bands. Now, based on our simple N=4 contiguous bits based band mapping, combining A and C will require us to move the start bit for address to band mapping to bit position 3 and this newly formed band (0**000**_**1**XXX) will also include band B. That means increasing the band range (or combining bands) may group multiple read-only and read-write bands together, and the newly formed read-write band cannot provide caching benefits to the read-only addresses in that band. In the above scenario, a multiband optimization can simultaneously grant epochs to bands A and C, and still preserve the read-only band B retaining the caching benefits.

The implementation details and the implementation cost of the multi-band optimization can be found in Section 7.1.5. The number of concurrent bands needed is application

dependent. We evaluated a multi-band optimization with 4 concurrent bands and that performed well across all the evaluated benchmarks. More details about the evaluation of the multi-band optimization can be found in Section 7.3.1.

4.2 Entangled Coherence

The STC rules introduced in Section 3.1 block writes that do not belong to the current epoch. This will incur heavy store completion latency and will lead to performance degradation in latency-sensitive processors. Further, the STC's epoch rule 3 (ER3) forces all processors to be in the same epoch at any logical time. This rule forces even processors that are writing to private data structures to wait for the correct epoch before issuing a store request. Ideally, only those processors that are sharing data should be in the same epoch. For example, in a quad-processor system, if processors 0 and 1 are sharing data among themselves, and processors 2 and 3 are not sharing any data, then both processor 0 and 1 should be in the same epoch but processor 2 can be in its own epoch and processor 3 can be in a third epoch. Such coexistence of multiple epochs can allow processors to retire stores without being blocked on an unnecessary common epoch. We define a new coherence mechanism called Entangled Coherence that allows multiple epochs to coexist.

4.2.1 Introduction to Entangled Coherence

Without loss of generality, we introduce the basic philosophy of entangled coherence (EntC) with two processors, a modifier and an observer. EntC observes that providing coherence between two processors essentially maintains these two processors in an entangled state with any state change to the modifier eventually made visible to the observer. A state change happens when the modifier modifies an address. EntC simply views coherence as a mechanism to maintain this entanglement after a state modification.[1] The order in which

[1]The above mentioned property is remarkably similar to the *quantum entanglement* phenomenon where a state change in one of the entangled particles affects the other. Hence the name entangled coherence.

P1	P2	P3	P4
1. Load A1	1. Load A2	1. Store D3	1. Store E1
2. Load B1	2. Load B1	2. Load D1	2. Load F2
3. Store A3	3. Store C3	3. Load D2	3. Load E2
4. Store C3	4. Load A3	4. Load D2	

Figure 4.2: Demonstration of entangled group formation. Processors accessing the same bands form an entangled group thus allowing epochs of those bands to be locally synchronized within that entangled group.

these state changes should be made visible to the observer is dictated by the consistency model. While SC does not allow a modifier to see its own state change without making it visible to the observer, TSO permits that possibility. Thus with TSO, the entanglement is not broken even if the observer sees a different value than the modifier.

4.2.2 Establishing Entanglement

Entangled Groups

Entangled groups allow multiple epochs to coexist at the same time by allowing a subset of processors to have their own common epoch. Figure 4.2 demonstrates a scenario where multiple epochs can coexist. In that figure, addresses belonging to the band corresponding to epoch_A (shown as A1, A2 and A3) and addresses belonging to the band corresponding to epoch_B are accessed by only processors P1 and P2. Hence, for maintaining coherence, epochs A and B need to be only synchronized with these two cores. Similarly, processors P3 and P4 are not sharing any data and they can be in their own epochs epoch_D and $\text{epochs}_{E,F}$ respectively without maintaining synchronization with the epochs of processors P1 and P2.

The processors that need to be synchronized together forms an entangled group (EG). For example, processors P1 and P2 are sharing data via $\text{bands}_{A,B,C}$ and hence their corresponding $\text{epochs}_{A,B,C}$ must be synchronized among them for their correct operation. Hence, P1 and P2 forms an EG via $\text{bands}_{A,B,C}$. The properties of EGs are given below:

1. A processor must be a member of at least one EG.
2. A band and its corresponding epoch can be assigned only to a single EG.
3. All processors accessing a band should be in the same EG.
4. Multiple bands (and corresponding epochs) can be assigned to an EG.
5. All members of an EG should be synchronized to the same epoch (or epochs) at any logical time.
6. Each EG should have at least one epoch assigned to it.
7. Each EG should have at least one member.

Applying the above mentioned properties to the example in Figure 4.2, processors P1 and P2 must be in the same $\text{EG}_{P1,P2}$ with $\text{epochs}_{A,B,C}$ assigned to that EG. Processor P3 can be on its own EG_{P3} with epoch_D assigned to it whereas processor P4 can be the sole member of a third EG_{P4} with $\text{epochs}_{E,F}$ assigned to it. EntC provides coherence by synchronizing epochs assigned to an EG across its member processors. For example, $\text{EG}_{P1,P2}$ toggles between epoch_A, epoch_B and epoch_C, and synchronizes it only between P1 and P2. Processors P3 and P4 are operating on private data and having their own EG allows them to retire their stores instantly without waiting on the correct epoch to arrive. Thus, allowing coexistence of multiple epochs with entangled groups reduces the store retirement delay from waiting on a common epoch.

It should be noted that a processor can be a member of multiple EGs. For example, if processor P2 was accessing band_D in addition to $\text{bands}_{A,B,C}$, then P2 and P3 would have formed an $\text{EG}_{P2,P3}$ via band_D. In that scenario, P2 will be a member of both $\text{EG}_{P1,P2}$ (via $\text{bands}_{A,B,C}$) and $\text{EG}_{P2,P3}$ (via band_D).

Quasi-entanglement: While the processors of an EG are strictly entangled by the epoch synchronization mechanism, the processors across EGs are in a quasi-entangled state. This quasi-entangled state is broken if a processor access a band whose epoch is not assigned

to its EG. Thus, for accessing addresses outside their currently assigned epochs, a processor should first become a member of the EG that owns that epoch. A processor that accesses multiple epochs may first need to merge its current EG with a different EG so that the merge will create a new EG that have epochs from both merged groups. In the degenerate case, all EGs will be merged together to form a single EG and its operation will be same as STC.

4.2.3 Entangled Coherence vs. Spatiotemporal Coherence

EntC follows the epoch rules (ER) (section 3.1) within an EG. The only exception is ER3 which states that all processors should be in the same epoch. Hence, it may seem that the localized synchronization operation of an EG violates ER3. However, the quasi-entanglement property of EntC ensures the bands/epochs assigned to an EG are not accessed by the processors outside it. Thus, although epochs are not explicitly synchronized across all processors with EntC, the effect is the same as STC. Thus EntC can be viewed as an optimization of STC that synchronizes epochs only among the CUs that access the bands corresponding to those epochs. Alternately, STC can be considered a special case of EntC with just one EG having all processors in it and all epochs assigned to it.

EntC provides performance benefits over STC in the following ways:

1. Only the producers and consumers of a band take part in an epoch synchronization enabling the synchronization operation to be localized to few processors thus lowering the epoch synchronization overhead.

2. The scratch space or private space used by a processor and not shared with other processors are filtered out as private bands and the private bands' epochs are exempted from epoch rules. Thus a processor can issue accesses to its private band without any epoch synchronization overhead and without incurring epoch-induced invalidation penalty.

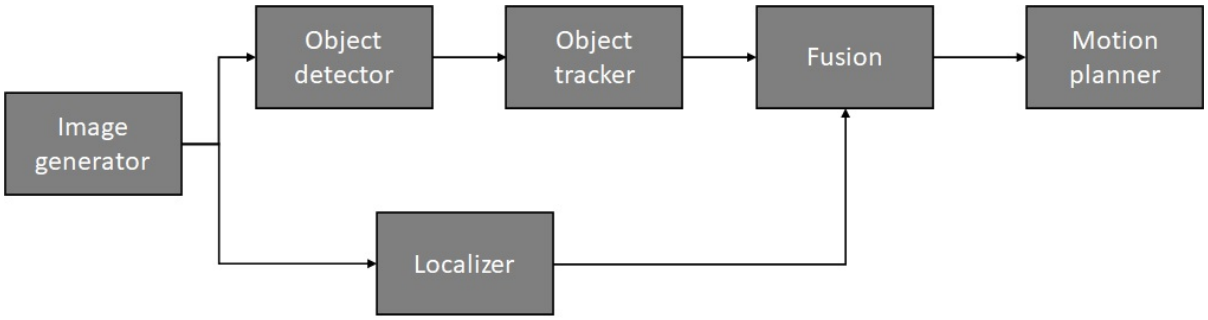


Figure 4.3: Autonomous driving pipeline. The data sharing pattern is fixed throughout the life cycle of this application and each stage of the pipeline shares data only with few other stages. Both these features can benefit from Entangled Coherence. More details about individual stages of this pipeline can be found elsewhere [68].

Multi-band Epoch vs. Entangled Coherence

Multi-band epoch optimization grants epochs of multiple bands simultaneously thus allowing multiple epochs to co-exist in physical/logical time just like EntC. However multi-band optimization grants epochs to bands that have pending writes to it and synchronizes these epochs across all processors. Entangled Coherence is more clever in its synchronization. It knows the sharing pattern and hence enables local synchronization among the consumers and producers of a band. Knowing the producers and consumers of a band also allows for private epoch optimization as discussed earlier. Both of these optimizations are not possible with multi-band. Additionally, since each EG implements STC among its members, the STC's multi-band optimization can be implemented within an EG further increasing its performance potential.

4.2.4 Applicability of Entangled Coherence

EntC works by grouping processors that share data into an EG. As such, it has the maximum impact on applications that have a fixed sharing pattern or have long phases with fixed sharing pattern. A class of applications that fit this perfectly is applications with software data pipelines such as autonomous driving [68].

A typical autonomous driving pipeline is discussed by Lin *et al.* [68] and summarized in Figure 4.3. The producers and consumers of each stage of the pipeline are fixed, and hence the sharing pattern is constant throughout the lifetime of the application. Additionally, each pipeline stage only shares its data with only few other stages and such a sharing pattern benefits from the localized synchronization of EGs. Because of these reasons, we use this autonomous driving pipeline as a use case for evaluating EntC (more details can be found in Section 6.3.3).

4.3 Chapter Summary

This chapter introduced four optimizations: (a) the epoch skipping optimization that enables faster write completion by providing epochs only to the bands with pending writes (Section 4.1.1), (b) the adaptive band optimization that identifies and filters out read-only bands from coherence overheads (Section 4.1.2), (c) the multi-band optimization that enables faster write completion by granting multiple bands simultaneously (Section 4.1.3), and (d) Entangled Coherence that localizes the epoch synchronization to a subset of processors by learning their data sharing pattern (Section 4.2).

In the next chapter (Chapter 5), we will define and formalize the Lockstep SC consistency model for GPUs.

Chapter 5

Consistency Model for Lockstep Execution

In the previous chapters we proposed solutions for coherence that are suitable for GPUs. In this chapter we transition to a related problem: that of consistency.

GPUs employ a single instruction multiple thread (SIMT) execution model in which multiple threads share a common instruction stream and execute instructions in a lockstep fashion. While strong models like SC and TSO have well-defined memory ordering rules based on the program order for CPU like threads [99], lockstep executing threads share the same program order. Hence it is unclear if the ordering rules must be applied to individual threads or should be common for all lockstep executing threads of a wavefront in a GPU. Towards this, in this chapter, we first argue that the program order should be defined for a wavefront in a GPU (Section 5.1). Then we formally define Lockstep SC consistency model that extend SC's ordering rules to a group of threads executing in lockstep (Section 5.2). Finally, we discuss both the programming and hardware design implications of this new consistency model (Section 5.3) and conclude the chapter with a summary section (Section 5.4).

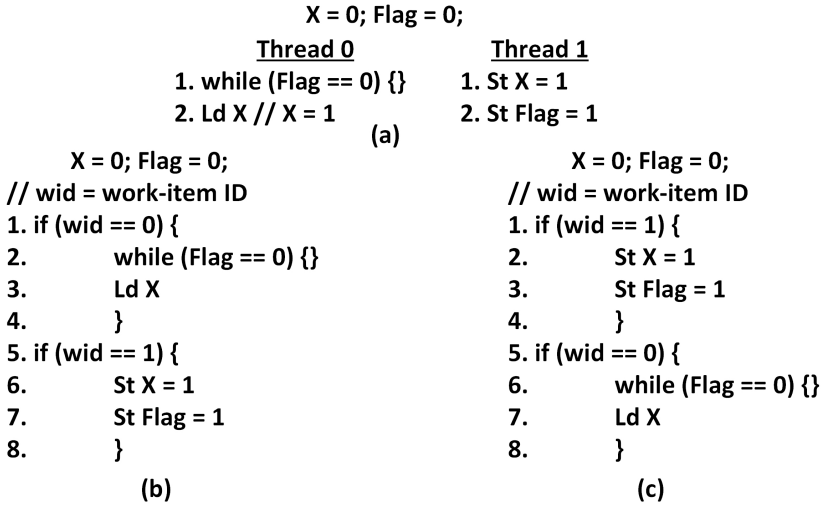


Figure 5.1: GPU program order. This figure demonstrates why program order should be defined for a wavefront in a GPU. The first code (a) shows two CPU like threads synchronizing via a *Flag* variable. The second code (b) shows two work-items of the same wavefront attempting the same synchronization. However, since both work-items share the same program order, this may result in deadlock. The third code (c) shows the code rewritten assuming wavefront program order.

5.1 Program Order for Lockstep Execution

In a CPU, the program order (PO) is defined for individual threads. However, since a GPU executes work-items of a wavefront in lockstep fashion, it is unclear if the PO should be defined for individual work-items or for the lockstep-executing wavefront. To understand the subtle difference, consider the synchronizing code in Figure 5.1(a). Here, *thread 0* and *thread 1* have their own PO and they synchronize with the *Flag* synchronization variable enabling the load of *thread 0* to see the value updated by *thread 1*. However, if the same synchronization is attempted between two work-items of the same wavefront, the assumption of them having independent PO can lead a programmer to write the code in Figure 5.1(b). Since the threads of a wavefront execute in lockstep, the *thread 0* will spin on the *Flag* variable without reaching the release operation by *thread 1* resulting in a deadlock in current commercially-available GPU architectures. The reason for this deadlock is the incorrect

assumption of independent PO for work-items within a wavefront. Independent PO based reasoning does not work with lockstep execution. Additionally, the same code may avoid deadlock on some GPU architectures with dynamic wavefront creation capability proposed in the literature [38]. This capability enables some work-items to execute a different basic block than its peers. This may sometimes avoid deadlock, which makes it even harder to find bugs and reason about correctness.

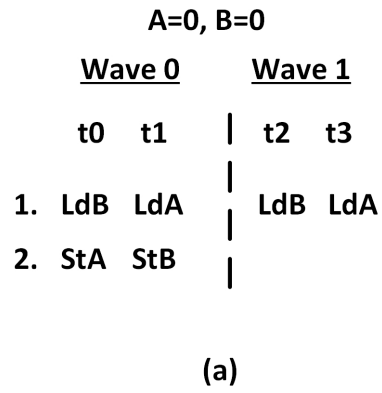
Furthermore, modern GPUs have scalar cores that execute scalar instructions of a wavefront [13, 14, 47]. The scalar instructions perform operations common to a wavefront, such as handling control flow. These scalar instructions are inserted by the compiler and their correct execution relies on a wavefront PO. It is also important to note that the scalar instructions cannot be even defined within the scope of a work-item PO.

Because of these reasons, we argue that the PO should be defined for a wavefront in a GPU. Figure 5.1(c) shows the code rewritten with the assumption of a per-wavefront PO. This code will not deadlock on the current architectures and the synchronization mechanism is easy to reason about.

5.2 Lockstep SC

SC ordering rules are defined for individual threads [99]. But GPUs execute work-items in lockstep and ordering memory operations from a lockstep executed instruction is unnecessary. Towards this, we propose Lockstep SC (LSC) that allows threads executing in lockstep in a wavefront to share the SC ordering rules.

For defining LSC, we use the same naming convention used in the Primer [99]: op1 $<_p$ op2 implies that op1 precedes op2 in that core's PO and op1 $<_m$ op2 implies that op1 precedes op2 in global memory order. To formalize LSC, in addition to $<_p$ and $<_m$, we define $=_p$ and $<_m>$. op1 $=_p$ op2 implies that op1 and op2 are logically executed simultaneously in the program order and op1 $<_m>$ op2 implies op1 and op2 are unordered in global memory



(a)

models/ outcomes	(t0,t1)	(t2,t3)
SC	(A=0,B=0) (A=1,B=0) (A=0,B=1)	(B=0,A=0) (B=1,A=0) (B=0,A=1) (B=1,A=1)
LSC	(A=0,B=0)	(B=0,A=0) (B=1,A=0) (B=0,A=1) (B=1,A=1)
SLSC	(A=0,B=0)	(B=0,A=0) (B=1,A=1)

(b)

Figure 5.2: *Extending SC to GPU* candidate models. (a) lists the memory accesses from two wavefronts wave 0 and wave 1. (b) lists the valid outcomes for different models. The differences in their outcomes underline that a phrase like *extending SC to GPU* is too vague for a consistency model and there must be a formal definition.

order. MOp(X) denotes a memory operation that reads and/or modify the state of address X in memory. Any execution is LSC if it adheres to the following conditions:

- If MOp(a) $<_p$ MOp(b) $\Rightarrow$ MOp(a) $<_m$ Mop(b); $a == b$ or $a \neq b$
- If MOp(a) $=_p$ MOp(b) $\Rightarrow$ MOp(a) $<_m>$ Mop(b); $a == b$ or $a \neq b$
- Ld(a) = Value of MAX_m {St(a) | St(a) $<_m$ Ld(a)}

The first condition states that all older memory operations to same or different addresses ($a == b$ or $a \neq b$) in PO should be made visible to the memory before any younger memory operation(s). The third condition specifies that a load to a memory location a should see the value of the last store in memory order to that location. These two conditions are same as SC but LSC relaxes the ordering constraints on operations that are logically executed simultaneously by adding another condition (second condition). As such, memory requests

originating from the same wavefront instruction and logically executed simultaneously need not be ordered in LSC.

5.2.1 Lockstep SC model vs. *Extending SC to GPU* Models

LSC is a seemingly straightforward extension of SC. But there is clearly value in formalizing the model to prevent undefined behaviors, and the model is useful as it makes a strong memory model practical in GPUs. We demonstrate that a seemingly intuitive idea of *extending SC to GPUs* is really subtle and can result in many complex, non-intuitive behaviors. To explain this, we introduce a strictly lockstep SC (SLSC) model by changing the second condition added by LSC to have a strict ordering among lockstep memory requests as follows:

- If MOp(a) $=_p$ MOp(b) $\Rightarrow$ MOp(a) $=_m$ Mop(b); $a == b$ or $a \neq b$

With this change, we have three candidate models for SC on GPUs: (a) SC that treats each GPU thread as a separate independent entity, (b) LSC that completely relaxes the memory ordering among lockstep requests, and (c) SLSC that requires lockstep requests to be visible to memory atomically. To understand the subtleties, let us consider the execution of two wavefronts shown in Figure 5.2(a). Threads t0 and t1 belong to wavefront 0 and threads t2 and t3 belong to wavefront 1. Figure 5.2(b) shows the valid outcomes of SC, LSC and SLSC for the PO shown in Figure 5.2(a). Since SC treats GPU threads as independent entities, SC allows the outcomes (0,0), (1,0) and (0,1) for threads t0 and t1. However, LSC and SLSC only allow (0,0) outcome because they enforce ordering among lockstep executing instructions. Both SC and LSC have the same valid outcomes for threads t2 and t3. However, SLSC presents the stores from a lockstep execution as an atomic update to the memory and hence restricts the valid outcome to just (0,0) or (1,1).

We highlight a few non-intuitive behaviors from the previous example. Although LSC relaxed the ordering constraints for lockstep executing instructions, the valid outcomes are fewer with LSC than SC. LSC assigns program order to an entire wavefront as opposed

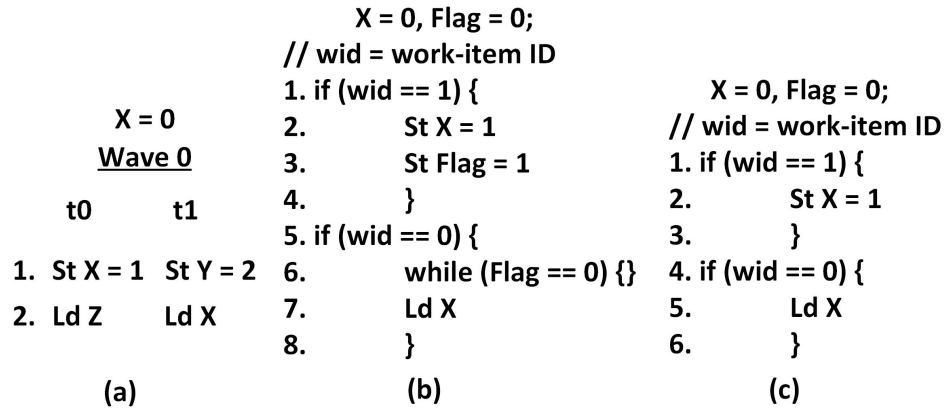


Figure 5.3: Implicit synchronization with Lockstep SC. If the two threads t0 and t1 of wave 0 in (a) are adhering to SC, then the load by t1 is not guaranteed to see the update to X. However, if these threads are part of the same wave, Lockstep SC guarantees Ld X=1. Similarly, the synchronization variable *Flag* in (b) is unnecessary with Lockstep SC because it provides implicit synchronization (because of the per-wavefront program order) to threads within the same wavefront and the code can be rewritten as in (c).

to individual work-items and with this wavefront PO, MOp(a) $<_p$ MOp(b) means that the memory operations from all lockstep executing threads are ordered by the PO thus reducing the number of valid outcomes. Additionally, naively extending SC to GPUs with the notion that *a wavefront in a GPU is equivalent to a thread in a CPU* will result in SLSC. Clearly, all these subtleties cannot be captured without a formal definition. In fact, the phrase *extending SC to GPUs* is too vague for one to even understand which among the three models (and possibly others not shown here) is actually implemented.

Figure 5.2(b) shows that SC does not provide a simple intuitive ordering for threads within a wavefront. Although SLSC provides a stricter global ordering both within and across wavefronts, implementing SLSC requires all wavefront memory accesses to complete atomically at the memory (or appear to complete atomically at the memory to an outside observer). This is prohibitively expensive. For example, if a lockstep execution is writing to multiple cachelines, the individual writes cannot be made visible until all writes are completed. Such an implementation will require locking the modified cachelines to prevent

an individual write from being visible to other cores till all writes from that lockstep execution are completed. A global ordering can also be achieved by forcing instructions executing in lockstep be consecutive within the per-wavefront PO and the same order to be exposed to all threads in the system creating a total order. This is more difficult to achieve than even SLSC. For example, this will require the lockstep executing operations to complete in some order, that is in some serial order (if grouped together, we will have to make the grouped operations appear atomically, which is SLSC), which in turn will force serialization of lockstep memory operations, taking away all the benefits of lockstep execution model of GPU. Hence, we chose LSC for our implementation. LSC is simple for programmers to reason about, and is agnostic to hardware or memory hierarchy organization unlike heterogeneous race free models (HRF) [54]. This makes an LSC program more portable than HRF and relieves a programmer from knowing the details of memory hierarchy organization of the targeted system.

5.2.2 Implicit Synchronization with Lockstep SC

Figure 5.3(a) shows two threads of the same wavefront executing a store followed by a load. Thread t1 is trying to read (Ld X) the store from thread t0 (St X). If the model is SC, since SC treats these two threads as independent entities, the Ld X is not guaranteed to see value 1. However, with LSC, the ordering constraints are applied across all threads in a wavefront and hence Ld X from t1 is guaranteed to see value 1. Figure 5.3(b) shows the code we discussed earlier in section 5.1 (Figure 5.1(c)). In the absence of LSC, the code needed a synchronization variable *Flag* to synchronize between two threads of the same wavefront. However, with LSC, the same code can be rewritten without any synchronization variable as shown in Figure 5.3(c). Since LSC defines program order to an entire wavefront and consequently its ordering rules are applied to all threads executing in lockstep, LSC provides synchronization without explicit synchronization flags for threads in the same wavefront.

5.2.3 Previous GPU SC Implementations

Singh *et al.* [96] proposed efficient SC implementation for GPUs by extending the work of Singh *et al.* [97] for CPUs. This work implemented SC for wavefront instructions (warp instructions) and argued that SC ordering need not be preserved across per-work-item (per-thread) instructions that execute in lockstep fashion. Similar implementations were proposed by other researchers as well but with timestamp based coherence protocols [87, 100]. Some of these implementations were in fact adhering to the LSC consistency model on GPUs [87, 96], not SC.

5.2.4 Interaction with the GPU Execution Model

LSC allows memory operations that are logically executed simultaneously in the program order to be unordered in memory. This definition makes it easy to reason about divergent executions in a GPU. A wavefront can diverge in a GPU when the work-items diverge in their conditional branch outcome. LSC's ordering relaxation to simultaneously executed operations then applies to only the subset of active work-items. A diverged execution can be simply treated as an execution phase in which only a subset of work-items issue memory accesses. These accesses still follow all LSC rules but the number of accesses that are ordered by $<_m>$ memory order will be less than the wavefront width.

GPUs with independent thread scheduling support [56] schedule threads independently. Consequently, there are no lockstep execution assumptions/guarantees in these GPUs [78] and LSC's ordering rule for lockstep execution is not applicable. Since LSC without lockstep execution rule is the same as SC, LSC naturally devolves to enforcing SC at work-item or thread level granularity with independent thread scheduling. Thus, LSC is well-defined for these GPU architectures as well. This property of LSC also makes it applicable to future SoCs featuring accelerators without the same lockstep execution model.

5.3 Implications

While SC defines the ordering rules for an individual CPU thread, LSC treats an entire wavefront as a single entity and defines the ordering rules for the entire wavefront. Although this difference may seem trivial, it has both programming and hardware design implications. The programming implications are further divided to programming implications for the application developer and for the low-level system software developer.

5.3.1 Implications for Application Development

We encourage the application developer to rely on underlying low-level software implementation to abstract out hardware specific details. For example, AMD GCN cross-lane operations [69] and CUDA warp-level intrinsics [91] expose hardware details like wavefront (or warp) size to the programmers. While an application programmer can write code with these intrinsics, those applications will not be portable across platforms with a different assumption about the wavefront size. However, with the low-level software abstracting out hardware details, the application writer can develop code without burdening themselves with the hardware specifics. With this envisioned model, synchronization primitives are also provided by the low-level software and hence the application developers need not concern themselves with the underlying consistency model. Thus LSC has limited impact on the application developer.

5.3.2 Implications for Low-level Software Development

Low-level software acts as the interface between application and hardware by providing hardware independent API calls and converting them to hardware dependent operations. For example, this layer can be used to implement common algorithms like shuffle or reduction using hardware dependent CUDA warp-level intrinsics [69] or AMD GCN cross-lane operations [91]. This layer also provides synchronization APIs that abstract out the behav-

ior of the underlying hardware consistency model. As such, LSC formalization impacts the programmability of this layer of software. For example, if synchronization is needed between two threads, with LSC, this layer can take advantage of the implicit synchronization flag if these threads are from the same wavefront. Naturally, this implementation needs to know the wavefront size of the underlying hardware to provide the correct implementation.

5.3.3 Hardware Design Implications

A GPU coalescer coalesces accesses to the same cacheline from a wavefront instruction into a single access [84]. Suppose a wavefront instruction is generating multiple stores but to the same address and the coalescer is coalescing these requests. If the stores are storing different values, the coalescer will have to select one store over the other to write to that address. The selection logic will be extremely complex for writes when one store partially overlaps with another. Since SC treats these threads as independent threads, a coalescer design adhering to SC will be forced to mimic an interleaving such that these stores are issued in some sequential order to the memory subsystem.

Figure 5.4(a) shows an example in which three partially-overlapping four-byte stores from the same wavefront instruction are issued by threads t0, t1 and t2 to byte offsets 0, 2 and 4 respectively. Figure 5.4(b) shows a valid SC interleaving in which the thread with a higher thread ID is treated as the younger store and thus overwrite the bytes written by older stores. From that figure, it can be seen that the final value comprises of bytes from all three stores. That means a coalescer adhering to SC will have to be implemented in such a way that there should be selection logic for each byte that is written. Prior work has shown that even address comparison to identify coalescable accesses is resource intensive [84]. The byte selection logic to determine the correct store values imposed by SC significantly adds to this overhead. LSC on the other hand allows these accesses to be unordered and thus does not impose such strict byte selection restriction on the coalescer design.

Irrespective of the actual order, assigning some order to the visibility of lockstep accesses

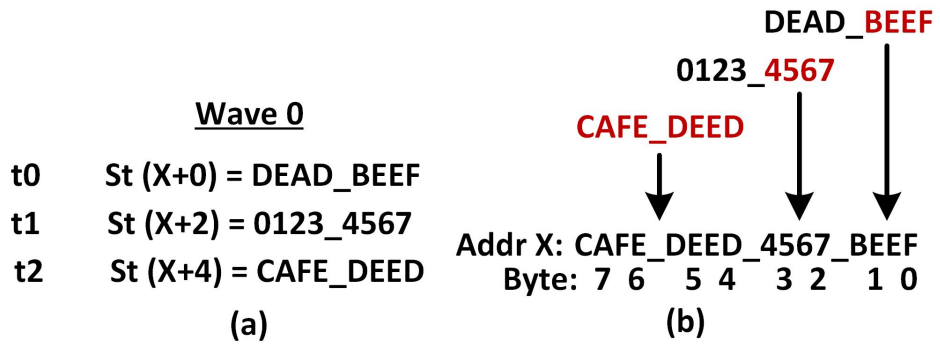


Figure 5.4: Coalescer adhering to SC. This figure shows the difficulty in designing a coalescer for a GPU with SC ordering rules. For a code with overlapping writes as shown in (a), the coalescer adhering to SC must have a selection logic for each byte because the final stored value should conform to an SC outcome even for memory accesses originating from lockstep execution. One such valid outcome is shown in (b). This selection logic increases the complexity of coalescer design. LSC avoids this design complexity by relaxing the ordering requirements of memory accesses originating from lockstep execution.

in the memory will complicate the coalescer design. LSC resolves this issue by making these accesses unordered ($<_m>$ memory order), thus facilitating a simple coalescer design. Specifically, if there are two or more stores to overlapping locations from a lockstep execution, a determination as to which one was the last store to that location is not possible and hence the value of the subsequent load to that location is undefined. This is similar to the reasoning provided in prior works where the outcome of racey accesses that "occur at the same time" are undefined [1, 25, 41, 54, 70]. As a result, LSC does not impose any restriction on the value of overlapping stores from a lockstep execution. Thus LSC facilitates simple coalescing logic. Additionally, since LSC implements a strong model, the coalesced requests from the same wavefront are to be issued in program order to the memory hierarchy and hence coalescing requests across wavefront instructions is not allowed. However, prior work has shown that coalescing across wavefront instructions is incredibly expensive and even existing designs do not implement it [84].

5.4 Chapter Summary and Outline of the Second Part of this Dissertation

We defined and formalized LSC in this chapter (Section 5.2). With this, we have filled the requirement for a formal definition of a strong model for GPUs. Section 5.3 of this chapter detailed the application development implications, the low-level software development implications and the hardware design implications of this new consistency model. The proposed LSC is simple for programmers to reason about and is portable across different memory hierarchy organizations because of its memory hierarchy agnostic nature.

This chapter concludes the first part of this dissertation. We start the second part with the experimental methodology chapter (Chapter 6). Chapters 7, 8 and 9 implements and evaluates Spatiotemporal Coherence, Entangled Coherence and Lockstep SC respectively. Chapter 10 concludes this dissertation.

Chapter 6

Experimental Methodology

In this chapter, we provide details of our simulation methodology for evaluating Spatiotemporal Coherence (Chapter 7), Entangled Coherence (Chapter 8) and an implementation of the Lockstep SC consistency model (Chapter 9). We will start with explaining our simulator (Section 6.1). Then we will explain the simulated GPU, APU and heterogeneous SoC systems (Section 6.2) followed by the workloads used for evaluation (Section 6.3).

6.1 Simulator

Simulator: We used the AMD gem5 APU simulator [47] that models a GCN GPU architecture [12]. This version of the simulator simulates the GCN3 ISA [13]. The simulator models an OoO cycle level CPU core. The memory subsystem is modeled in Ruby [73].

The simulation parameters are given in Table 6.1. We simulated a system with two dynamically scheduled x86 CPU cores and eight GCN compute units (CU). Each CU has a private L1 data cache. The L1 instruction cache is shared by four CUs. The CPU has a private L1 data cache and instruction cache backed by a private L2 cache. Both CPU cores share an L3 cache.

Table 6.1: Simulation parameters

CPU	2 x86 cores @ 3GHz
CPU Clock	3GHz
CPU Cores	2
ROB Size	192
CPU L1 Data Cache	64KB (2-way associative)
CPU L1 Instruction Cache	32KB (2-way associative)
CPU L2 Cache	2MB (8-way associative)
CPU Shared L3 Cache	8MB (16-way associative)
GPU	8 GCN CUs @ 1GHz
GPU Clock	1GHz
GPU CUs	8
GPU Exec Units	5 (4 VALU and 1 AGU)
GPU Exec Unit width	16
GPU L1 Data Cache	16KB/64KB (64-way associative)
GPU L1 Instruction Cache	16KB (16-way associative)
GPU Shared L2 Cache	512KB (16-way associative)

6.2 Simulated System

Spatiotemporal Coherence protocol is a natural fit for latency tolerant accelerators such as a GPU, and hence we evaluate it on a GPU system. Lockstep SC is a consistency model for GPUs but the model naturally devolves to SC for CPU threads and hence applicable to both CPU and GPU. So we evaluate it on an accelerated processing unit (APU) SoC. Entangled Coherence protocol is suited for latency sensitive applications but with a fixed sharing pattern such as a mobile SoC [52] or autonomous driving pipeline [68]. Hence we evaluate it on a heterogeneous SoC system.

6.2.1 GPU System

We simulated a GPU with 8 compute units (CU). Each CU has 4 vector ALUs (VALUs) that share a common front end. A VALU along with its instruction buffers and share of the register file is called a SIMD unit. The instruction buffer of each SIMD unit can accommodate ten wavefronts (or warp). A wavefront is 64 workitems (or GPU thread) wide and takes 4 cycles to execute on a 16 wide VALU. There is an address generation unit (AGU)

in each CU to execute vector memory instructions. The AGU is also 16 wide and a wavefront takes 4 cycles to execute a vector memory instruction. The wavefront wide instructions such as branch instructions and waitcount instructions are executed by a scalar core [47]. This architecture is similar to many commercially available GCN-based GPUs [27, 62].

A virtual address generated by the AGU is translated by a TLB before sending it to a physically indexed and physically tagged private L1 data cache. Since the AGU can generate up to 16 requests in a single cycle, these requests are coalesced by a TLB coalescing unit before accessing the TLB. This TLB coalescer coalesces multiple requests to the same page address reducing the GPU TLB access bandwidth. Such a design is proposed by Power *et al.* [81] and we adopted it in our baseline. The translated addresses go through one more layer of coalescing at the memory coalescing unit where physical addresses within a cache line boundary are coalesced together. This coalescing is necessary to reduce cache access bandwidth [84].

The GPU L2 cache is shared by all CUs but the write-through write-purge L1s are private to a CU. The baseline GPU system adheres to release consistency model. Naturally, these caches in the baseline are managed by software inserted release consistency synchronization markers and made coherent only at the synchronization boundaries [49, 54]. We evaluate Spatiotemporal Coherence along with its three optimizations (epoch skipping, adaptive band and multi-band epochs) against this GPU system.

6.2.2 APU System

An APU consists of a GPU and a CPU communicating over a shared address space. Figure 6.1 shows our baseline APU system with integrated CPU and GPU. The GPU is same as the one discussed earlier in Section 6.2.1. The dynamically scheduled out-of-order (OoO) CPU core has a split load/store queue design with separate load queue and store queue per thread. The reorder buffer (ROB) commits instructions in program order and the committed stores are taken out of store queue and moved into a store buffer. As such, the CPU core

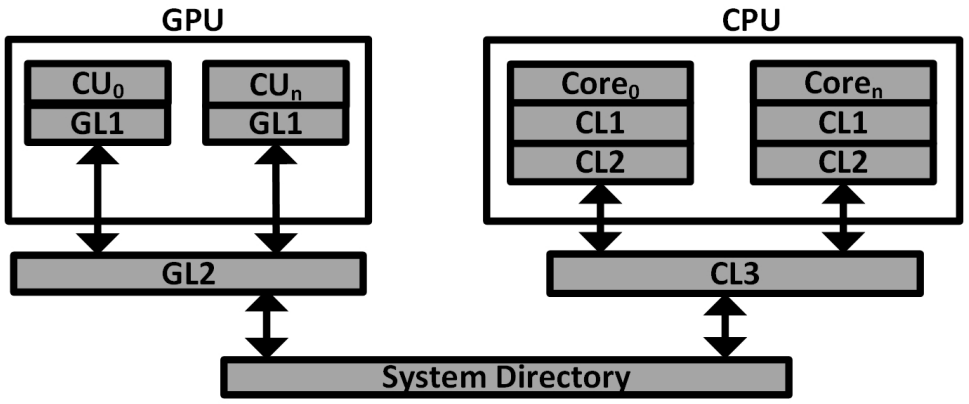


Figure 6.1: Baseline APU system. The CPU and GPU are coherently connected by a system directory. GPU has multiple CUs with private L1 caches (GL1) and a shared (GL2). The CPU has multiple cores with private L1 and L2 caches (CL1 and CL2 respectively), and a shared L3 (CL3).

adheres to total store order (TSO) memory model. The core supports speculation and relies on invalidation messages forwarded from the cache subsystem for detecting and correcting mis-speculated memory accesses. When a cacheline gets invalidated, the invalidation notification is forwarded to the core and the core replays any speculative load that has read that cacheline. For the same reason, an eviction notification is also forwarded to the core. This technique proposed by Gharachorloo *et al.* [42] allows the core to do aggressive memory access speculation.

The CUs and CPU cores communicate over a single shared address space with the help of coherent caches. The GPU L2 cache (GL2) is shared by all CUs but the L1s (GL1) are private to a CU. CPU cores have private L1 and L2 caches (CL1 and CL2) and a shared L3 (CL3). The private CL2 is inclusive of CL1 but shared caches in both CPU and GPU are non-inclusive.

The system directory acts as the ordering point between CPU and GPU coherence requests. It also hosts an inclusive probe filter that keeps a 2-bit sharer vector indicating if a cache block is in CPU caches or in GPU L2 or both. The probe filter also keeps track of the cache in ownership states [99]. CPU caches are write-back and implement a *read*

for ownership (RFO) protocol. This RFO protocol preserves the single writer invariant of cache coherence protocol [99]. The GPU caches are write-through and implement a simple VI (valid-invalid) coherence protocol. In our baseline implementation, invalidation messages generated by a write request are not forwarded to the GL1. The GL1 caches are made coherent by synchronization operations defined in the heterogeneous-race-free (HRF) memory model [54]. Consequently, the probe filter is inclusive of only GL2 cachelines. This baseline APU system is similar to many commercially available APUs [27, 62] and is used as baseline in many recent APU publications [11, 49, 80].

6.2.3 Heterogeneous SoC System

A heterogeneous SoC system typically consists of both general purpose processors, and specialized programmable and fixed function processing engines [52, 68]. Since modeling different types of processing engines is not a viable option, we are emulating a heterogeneous SoC system using a GPU with different processing engines represented by a compute unit in a GPU.

Our objective with this approach is not to emulate the SoC with cycle level accuracy. Our goal is to evaluate the memory system performance with Entangled Coherence for few software use cases on these SoCs. The processing engines in a heterogeneous SoC often run under strict latency constraints [68], and hence we modified the GPU parameters listed in Table 6.2 to make the compute units latency sensitive. Other than the differences listed in that table, the GPU system is same as the one discussed earlier in section 6.2.1.

Table 6.2: Changed GPU parameters

GPU Clock	1 to 2 GHz
GPU VALUs	4 to 1 VALU
GPU Outstanding Memory Instructions	4 to 1 instructions
GPU L1 Latency	20 to 4 GPU cycles roundtrip
GPU L2 Latency	160 to 20 GPU cycles roundtrip
GPU Memory Latency	260 GPU cycles roundtrip

Table 6.3: Workloads

AMD Compute Apps [15]
SN application proxy (snapc): Particle transport
High-Performance geometric multi-grid (hpgmg): Linear solver
Classic molecular dynamics (comd) : Molecular dynamics simulation
Livermore unstructured Lagrangian explicit shock hydrodynamics (lulesh) : Hydrodynamics calculation
HCC Example App [18]
Array bandwidth (array-bw): Bandwidth stress test
Fast fourier transform (fft): Fourier transform
Molecular dynamics (md): Particle movement
Sparse matrix-vector multiplication (spmv): Matrix-vector multiplication
Double-precision GEMM (dgemm): Matrix multiplication
Rodinia [34]
Back propagation (backprop): Backpropagation of neural networks
K-nearest Neighbors (nn): Nearest neighbor classification
Breadth-first search (bfs): Breadth-first traversal/search for a graph/tree
Pannotia [33]
Maximal independent set (mis): Graph partitioning
Floyd-Warshall (fw): Shortest path
Betweenness centrality (bc) : Vertices' centrality

6.3 Workloads

We used benchmarks from AMD compute app suite [15], HCC example app suite [18], Rodinia benchmark suite [34] and Pannotia benchmark suites [33] to evaluate Spatiotemporal Coherence and Lockstep SC. The APU version of these benchmarks shares data between CPU and GPU via a shared virtual address space without any explicit data copy. The GPU phase of these benchmarks were used for evaluating Spatiotemporal Coherence.

The Rodinia and Pannotia benchmarks are written in HIP [19] but rest of the benchmarks are written in HCC [17]. Table 6.3 lists these benchmarks and provides their brief description[1]. In addition to these generic benchmarks, few handwritten benchmarks were used to evaluate certain scenarios not covered by the generic ones. Since Entangled Coherence is implemented on a heterogeneous SoC, we evaluated it exclusively with a hand written software pipeline generator application. Next, we delve into the details of these handwritten workloads.

[1]Not all the benchmarks listed in this table are used in the evaluation of both Spatiotemporal Coherence and Lockstep SC. The reasons being (a) HIP or HCC port of some benchmarks were not available at the time of experimentation, or (b) the simulator produced incorrect execution and the results were discarded

6.3.1 Handwritten Workloads for Spatiotemporal Coherence

We developed a fine-grain data sharing benchmark (fg-share) in which all workgroups attempt to enter a critical section, and then read and update a shared data structure in place, emulating a centralized ledger update. With the workgroups contending to enter into the mutually exclusive critical section implemented with atomic compare-and-swap, and with in-place updates, this benchmark significantly pressurizes the coherence mechanism. We also evaluated a cache-reuse benchmark that launches 10 kernels with each kernel reading the same read-only array and updating a second array. Additionally, a write-intensive vector copy (vec-cpy) benchmark that copies a buffer to another is also evaluated. Thus, we evaluate Spatiotemporal Coherence with irregular workloads from Pannotia suite [33](fw, mis, bc), for coarse-grain data sharing scenarios (snapc, fw, mis) and fine-grain data sharing (bc, fg-share), under pathological scenarios for STC (fg-share, vec-cpy), and for regular GPU-friendly workloads (remaining benchmarks).

6.3.2 Handwritten Workloads for Lockstep SC

We developed a double precision matrix multiplication application (dgemm) to evaluate Lockstep SC because this application was reported to perform poorly with strong consistency models in a prior study [96]. Additionally, we ported two benchmarks to HIP [19]; *hashtable* and *ATM*; from Fung *et al.* [39] to demonstrate the performance advantage of a strong model for fine-grained synchronizing and data sharing GPU applications. Each GPU thread in *hashtable* benchmark inserts a key-value pair to a hash table slot necessitating fine-grained synchronization of threads trying to insert value to the same slot and fine-grained data sharing to get the most updated copy of the hash table after a new element is inserted. The *ATM* benchmark simulates GPU threads doing transactions between two accounts and similar to *hashtable*, requires fine-grained synchronization and fine-grained data sharing between GPU threads. We only simulate the GPU phase of these two benchmarks. Table 6.4 lists the benchmarks used for evaluating Lockstep SC. That table also provides

the amount of data (in MB) coherently shared between CPU and GPU by each benchmark. This coherently shared data as a fraction of the total data accessed by GPU (in %) is also provided.

Table 6.4: APU benchmarks

AMD Compute App	**Description**	Data shared between CPU and GPU (in MB and in %)
comd	Molecular dynamics simulation	80.14 (11.24%)
lulesh	Hydrodynamics calculation	64.73 (74.84%)
hpgmg	Linear solvers	17.43 (42.24%)
HCC Example App		
array	Streaming write to an array	1.64 (6.36%)
bitonic	Bitonic sort	415.63 (66.68%)
fft	Fast Fourier transform	17.16 (96.69%)
spmv	Sparse matrix-vector mulitply	6.24 (16.32%)
Rodinia		
backprop	Back propogation	3.09 (61.54%)
bfs	Breadth first search	27.26 (33.19%)
nn	Nearest neighbour	1.3 (44.47%)
Hand written		
dgemm	Double precision matrix multiplication	3.13 (26.86%)

6.3.3 Software Pipeline Generator for Entangled Coherence

There are diverse set of heterogeneous SoC systems targeting different use cases. It is impractical to model and simulate each of these systems for these different use cases. Instead, we developed a software pipeline generator that can be used to emulate some of the fundamental characteristics of the software running on the processing elements of these heterogeneous SoCs. This software pipeline generator emulates injecting traffic from various processing elements to a shared memory address space and thus can be used for studying the impact of memory system performance of a coherence implementation such as Entangled Coherence.

The rationale behind using such a pipeline generator can be demonstrated with an autonomous driving pipeline discussed by Lin *et al.* [68] and summarized in Figure 6.2. The image generator stage feeds the images to the object detector and localization module. The object detector processes the image and sends its output to the object tracker. The fusion engine combines the objects tracked by the trackers with the vehicle location information provided by the localization engine. This combined information is then fed into the mo-

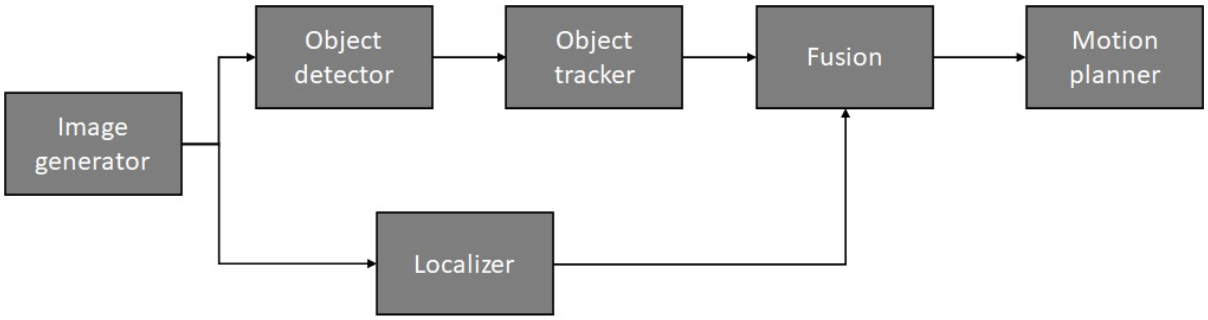


Figure 6.2: Pipeline generator use case: autonomous driving pipeline. This pipeline presents a use case in which the number of producers and consumers vary across different pipeline stages.

tion planner which dictates the next vehicle action. More details about each stage of this autonomous driving pipeline can be found elsewhere [68].

Each processing element in the autonomous driving pipeline is tasked with one operation. Irrespective of the exact operation performed, the memory accesses of those operations can be abstracted to 4 phases:

1. Read phase: frame(s) is(are) read from the previous pipeline stage(s).
2. Reuse phase: the read frame(s) is(are) optionally reused while doing the processing in which case there can be cache hits.
3. Scratch phase: the algorithm may optionally use a private memory region as a scratch-pad memory for performing the computation.
4. Write phase: the processed output frame(s) is produced for the next stage(s).

Thus, from memory access and memory system performance standpoint, each stage of this pipeline can be abstracted to a *pipeline stage* that performs the 4 phases discussed above. Our software pipeline generator models such an abstract pipeline stage.

The pseudo code for the abstract pipeline stage is shown in Figure 6.3. We are modeling a double buffered pipeline so that the producer stages of the pipeline are not stalled frequently because of the non-availability of buffer space while consumer stage is processing data. The lines:5,6 of the pseudo code shows the selection of the current buffer from the two available

```
1. myCurrentFrame = 0;
2. while(myCurrentFrame < maxFrames) {
3.     prod = cons = 0;
4.    // double buffered pipeline, choose A or B frame
5.    currentRdFrame = (myCurrentFrame%2 == 0) ? A-RdFrame : B-RdFrame;
6.    currentWrFrame = (myCurrentFrame%2 == 0) ? A-WrFrame : B-WrFrame;
7.    // wait till producer(s) produces the next frame
8.    while(prod < numProducers) {
9.            while(producerFrame[prod] == myCurrentFrame);
10.           prod++;
11.    }
12.   // make sure consumer has consumed the previously produced frame
13.   // "-2" because double buffering permits one frame to be buffered safely
14.   while(cons < numConsumers) {
15.           while(consumerFrame[cons] == myCurrentFrame-2);
16.           cons++;
17.   }
18.   // read new frame from producer
19.   readFrame(currentRdFrame);
20.    // if there is reuse, read the read frame. Simulates read cache hits.
21.   while(reuseSize > 0) {
22.           reuseReadFrame(currentRdFrame);
23.           reuseSize--;
24.   }
25.   // if there is scratch, perform private scratch accesses
26.    while(scratchUseSize > 0) {
27.           writeToScratch();
28.           readFromScratch ();
29.           scratchUseSize--;
30.    }
31.     // produce new frame to the consumer
32.    writeFrame(currentWrFrame);
33.    // increment the frame number so that both producer(s) and consumer(s)
34.    // knows which frame this stage is processing
35.    myCurrentFrame++;
36.}
```

Figure 6.3: Pseudocode for an abstract pipeline stage. Many diverse heterogeneous SoC applications can be emulated by creating pipeline stages with different read and write frame sizes, different reuse and scratch sizes from this abstract model, and then connecting them together.

buffers. Lines:8-11 shows *this* stage waiting for its producer(s) whereas lines:14-17 shows it waiting for its consumer(s). These *waiting* ensure producers have produced the data needed for the current frame and the consumers have finished consuming data produced by the previous frames so that the write buffer can be overwritten. Line:19 performs the read phase, lines:21-24 performs the reuse phase, lines:26-30 performs the scratch phase, and line:32 performs the write phase. Finally, the current frame is updated (line:35) so that both the producer(s) and consumer(s) of this pipeline stage know the frame number of their consumer and producer respectively.

We implemented this pseudocode on a GPU with one pipeline stage (representing a processing engine in an SoC) implemented as a wavefront. Only one wavefront is scheduled to a CU in our implementation. This is achieved by instantiating as many CUs as the number of wavefronts - i.e. pipeline stages - in the emulated pipeline. Since our baseline GPU caches are managed by release consistency synchronization markers, the waiting on consumer and producer are performed with load acquires, and the current frame are updated by a store release operation. Many of the applications running on heterogeneous SoCs can be emulated with this pipeline by instantiating a processing element with a pipeline stage with proper frame size, reuse size and scratch size, and then connecting these pipeline stages creating the desired producer-consumer relationships between the processing elements.

We use a linear 5-stage pipeline (*linear*) for the evaluation of Entangled Coherence. The read and write frame size of this linear pipeline is fixed to 128KB. We also emulate the autonomous pipeline (*autonomous*) shown in Figure 6.2 and evaluate the Entangled Coherence with it. We are assuming the images are produced by 5 LIDAR sensors and 1 camera producing 75 MBps [51]. At a frame rate of 60 frames per second, 75MBps data rate equates to a frame size of 1.25MB. We are approximating this to a 1MB frame for our autonomous pipeline model and hence fixing our read frame and write frame size to 1 MB for all its pipeline stages. Although all the pipeline stages will not have the same frame rate for a real application, a pipeline's performance is typically dictated by its slowest stage and

from the coherent data sharing perspective, the slowest pipeline is the image data transfer pipeline. Hence we use the slowest stage's frame size for all pipeline stages. We will use the two pipelines *linear* and *autonomous* with various reuse and scratch sizes to evaluate the Entangled Coherence protocol.

6.4 Chapter Summary

This chapter presented our simulator (Section 6.1) and provided details about the GPU, SoC and APU systems we will be using for our evaluations (Section 6.2). We also discussed the workloads in Section 6.3.

The next chapter implements and evaluates Spatiotemporal Coherence on a GPU system.

Chapter 7

Spatiotemporal Coherence for GPUs

We first discuss the implementation of Spatiotemporal Coherence (STC) on a GPU in this chapter (Section 7.1). Then we will demonstrate the correctness of the implementation with a working example (Section 7.1). Finally we conclude this chapter with an evaluation of our implementation (Section 7.3) followed by a chapter summary (Section 7.4).

7.1 Implementation

We implement STC on a GPU with per compute unit (CU) write-through private L1 caches and a shared L2 cache as shown in Figure 7.1. The GPU adheres to the release consistency (RC) model and implements a VI (valid/invalid) protocol for making L1s coherent at RC synchronization points [11, 12, 49, 60, 77, 80, 101]. We start with the RC implementation on our baseline GPU followed by our baseline STC implementation and proposed optimizations. This will be followed by our baseline STC implementation without any performance optimizations, and then add the epoch skipping, adaptive band and multi-band epoch optimizations to that baseline.

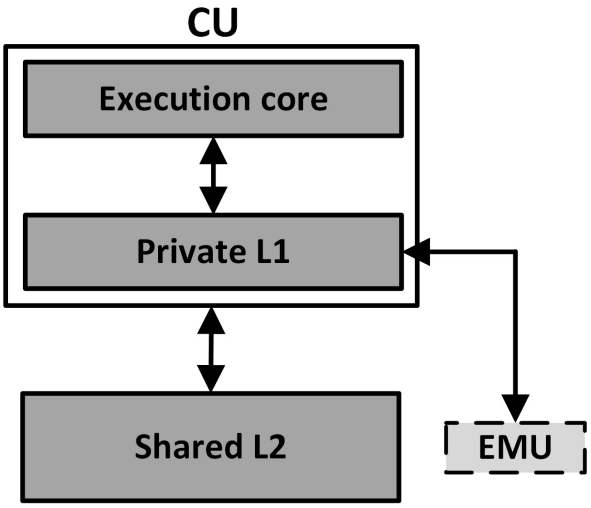


Figure 7.1: Spatiotemporal Coherence baseline with epoch management unit. The epoch management unit wakes up at fixed time interval and synchronizes the new epoch across all the CUs.

7.1.1 Release Consistency Implementation on Baseline GPU

Our baseline GPU executes GCN3 ISA [13] and implements RC with the two instructions: (a) *s_waitcnt* and (b) *buffer_wbinvl1_vol* [13, 14]. The *s_waitcnt vmcnt(0)* stalls the wavefront till all pending memory operations are completed and is used for ensuring the ordering requirements of acquire/release operations in the GCN3 based GPUs. To track the pending stores, the store completion acknowledgments are forwarded to the execution pipeline. The pending loads are simply tracked by load data returns. The *buffer_wbinvl1_vol* instruction performs the L1 cache invalidation part of the acquire operation. The baseline flash invalidates all valid L1 entries in one cycle by flipping the valid bits. An acquire-release operation is implemented by a combination of these two instructions. A more detailed discussion of the acquire-release operations with GCN3 ISA can be found elsewhere [13, 14].

Our STC implementation adheres to the RC model. The release operation is implemented similar to the baseline GPU with the write completion acknowledgments forwarded to the GPU pipeline and the *s_waitcnt vmcnt(0)* instruction stalling the wavefront till the write

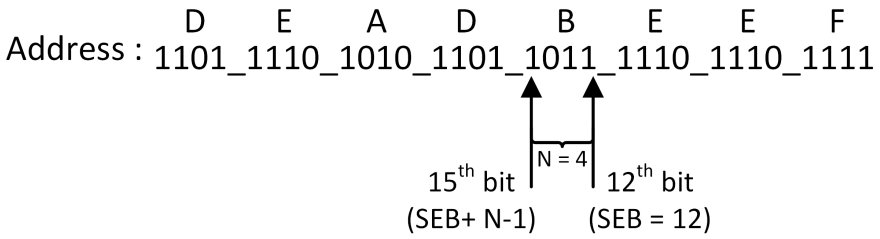


Figure 7.2: Epoch and band implementation. We use N contiguous bits of an address to identify a band, with matching address bits in that set of N bits forming a band. In this figure, N = 4 and the start epoch bit (SEB) = 12. Consequently, this address belongs to band 0xB.

completion. However, the *buffer_wbinvl1_vol* instruction is treated as a no-op since STC private caches do not cache stale data. In addition to implementing release operations, the *s_waitcnt* instruction can be used for dependency resolution and extraction of instruction level parallelism by allowing wavefronts to execute beyond outstanding memory instructions. Since this is beyond the scope of this paper, interested readers can refer Gutierrez *et al.* [47] and the AMDGPU backend user guide [14] for more details.

7.1.2 Spatiotemporal Coherence Baseline

STC relies on two concepts: epochs that subdivide time, and bands that subdivide the address space. The potential design space for these is large, as STC will operate correctly for very flexible definitions of each, as long as the epoch rules are satisfied. In this section, as a starting point, we define both epochs and bands in the simplest possible way: as fixed-duration windows of time and as contiguous regions of the address space. More complex and flexible definitions of epochs and bands may provide additional advantages, but, as we will show, even the simplest definitions presented here enable the correct operation with high performance.

In our implementation, we use N contiguous bits of an address to identify a band, with matching address bits in that set of N bits forming a band. We then associate an epoch

in time for every such band. Thus, our implementation will have 2^N total epochs. The *start epoch bits* (SEB) identify the position from which these N contiguous bits should be extracted to form a band. This simple implementation guarantees that every band has one epoch conforming to the deadlock avoidance rule DAR1. Figure 7.2 shows the epoch and band with N=4 and SEB=12 for an address 0xDEAD_BEEF. Since the four bits from 12^{th} bit is 0xB (11 in decimal), this address corresponds to epoch_{11} and the set of addresses with bits[15:12] = 0xB forms band_{11}. Thus, this simple band implementation allows individual CUs to identify the band of a request from only its address for a given N and SEB.

A CU tracks the current epoch with an N-bit current epoch register (CER). For implementing the epoch rules, a CU compares the band of a memory request to the current epoch stored in CER. Only stores with bands matching the current epoch are issued to the cache and the remaining stores are blocked. All blocked stores are moved to a blocked store queue (BSQ) and they are reanalyzed on an epoch change. The BSQ is banked so that multiple stores can be analyzed in a cycle. The store data is kept in the coalescing buffers [65, 84] and only addresses of the blocked stores are moved to BSQ. If this coalescing buffer (or BSQ) gets full, the backpressure mechanism will stall the GPU pipeline, thus preventing the overflow of these buffers. The coalescing unit can only coalesce requests from the same wavefront instruction [84]. It also blocks requests to an address when there are older outstanding requests to that address. Thus, since the coalescing unit blocks loads under a pending write to the same address from being issued, the younger loads need not access the BSQ for reading the last store. The loads belonging to the current epoch's band are not allowed to cache and hence are issued as non-cacheable loads.

To synchronize epochs across all the CUs, we implemented a epoch management unit (EMU), which plays a conceptually simple role in STC: ensuring that all CUs transition across epochs correctly at well-defined epoch boundaries. The epochs are defined as a fixed duration of time in this implementation. More complex implementations can have a dynamic epoch duration, but, as we will show, even this fixed duration epoch provided good

performance for the benchmarks we evaluated. The EMU periodically wakes up at this fixed time intervals and changes the epoch in a round-robin fashion for all CUs. It employs an N-bit rollover epoch counter to track the current epoch. The wakeup logic of the EMU is implemented with a wakeup counter. This counter is incremented every clock cycle till the count reaches a predefined value. Thus, the wakeup counter is counting the number of EMU cycles and when that counter reaches the predefined number of cycles, the EMU is woken up. When the EMU wakes up, it increments the epoch counter and this incremented value is used as the new epoch. This mechanism represents the turn-based strategy for selecting new epoch.

Figure 7.1 shows the STC baseline after adding the EMU. For synchronizing the new epoch with all the CUs, the EMU uses a four-way handshake. This four-way handshake involves two epoch requests and the acks to these requests. Figure 7.3 illustrates this operation in detail. The EMU wakes up and broadcasts a *PrepareEpochChange* request to all the CUs①. Each CU then stops issuing stores, waits for the outstanding stores to complete, and responds back with the *ReadyAck* notifying the EMU that they are ready for an epoch change ②. The EMU on receiving ReadyAcks from all CUs broadcasts the *ChangeEpoch* request ③. The ChangeEpoch request carries the new epoch value and the CUs on receiving this ChangeEpoch request transition to the new epoch after responding with a *DoneAck* ④. The epoch transitioning ends after the EMU receives DoneAck from all CUs.

On receiving a PrepareEpochChange message, the CUs block all stores and wait for all outstanding stores to complete. Moving to a new epoch with pending stores will result in those stores modifying a band that does not belong to the current epoch violating ER2. When all outstanding stores are complete, the CU responds back to the EMU with a ReadyAck. Thus ReadyAck is a guarantee given by that CU to the EMU that no further modifications to any band will be carried out by that CU. Once EMU receives ReadyAcks from all CUs, EMU broadcasts the new epoch value to all the CUs with a ChangeEpoch message.

On receiving the ChangeEpoch message, the CUs update their CER with the new epoch

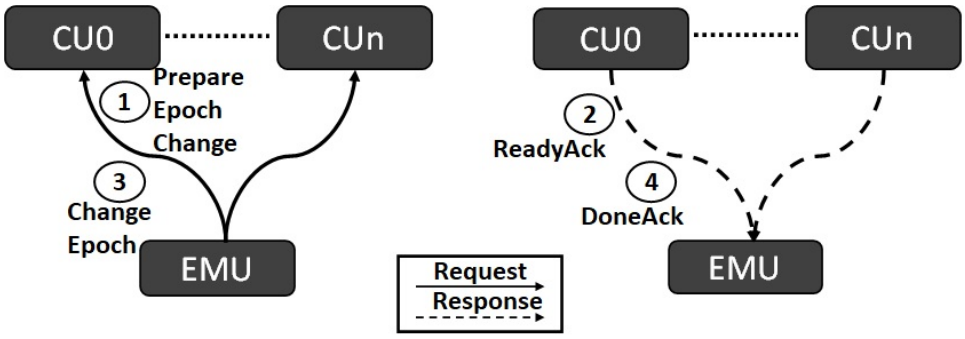


Figure 7.3: Epoch transition and synchronization. The epoch management unit synchronizes epochs with a 4-way handshake. The synchronization starts with a *PrepareEpochChange* request broadcasted to to all the CUs①. Each CU then stops issuing stores, waits for the outstanding stores to complete, and responds back with the *ReadyAck*②. After receiving ReadyAcks from all the CUs, epoch management unit broadcasts the *ChangeEpoch* request ③. The CUs on receiving this ChangeEpoch request transition to the new epoch and respond with a *DoneAck* ④.

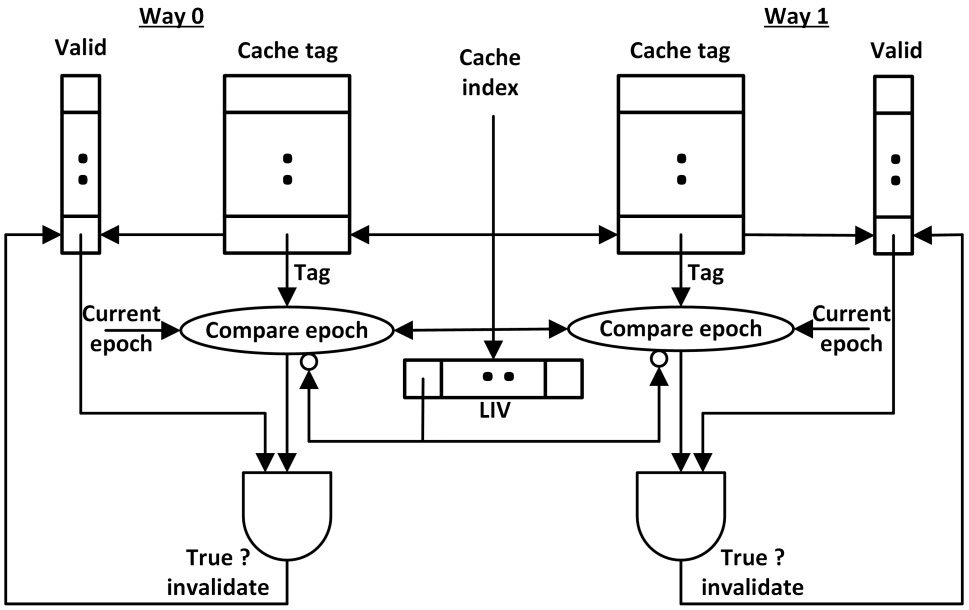


Figure 7.4: Lazy invalidation hardware. With lazy invalidation, every tag comparison initiated by a demand access is accompanied by an epoch comparison. If a way's entry matches the current epoch, that entry is invalidated adhering to epoch rules. The lazy invalidation vector (LIV) stores the cache sets that are already analyzed in this epoch and disables the epoch comparators for these sets thus preventing redundant comparisons.

from that message. The blocked stores in the BSQ are now reanalyzed and the stores to the new epoch's band are removed from BSQ and issued to the caches.

Lazy Invalidation

An epoch transition also requires valid entries that belong to the current epoch's band to be invalidated. To avoid a cache walk for performing this invalidation, a lazy invalidation mechanism is implemented that delays these invalidations till the first access to a cache set. Whenever a demand request reaches a cache set, the tags are read, and they are compared against the tag of the demand request. We leverage this normal cache operation to additionally compare the bands of the addresses stored in that cache set and if the bands match the current epoch, that entry is invalidated (only needs to be done once per cache set per epoch transition). To avoid redundant comparisons for an already analyzed cache set, the cache controller is augmented with a lazy invalidation vector (LIV). The LIV is implemented as a bit vector with 1-bit per cache set. All LIV entries are reset at the beginning of an epoch. The first demand access to a cache set sets its LIV entry. Once a LIV entry is set, further epoch comparisons for that cache set are disabled.

Figure 7.4 shows the lazy invalidation hardware for a 2-way cache. The demand request reads the tags and the corresponding LIV entry. If the LIV entry is not set, the bands of the addresses stored in the tags are compared against the current epoch. Since the cache index may also be needed to compute the band, both the tag and index are sent to the epoch comparator. If there are matches, then the valid bits of those tags are reset and the LIV entry is set. Since the LIV entry controls the enable bit of the epoch comparators, the comparators are disabled for further epoch comparisons to that set. The demand request already reads the tag array for normal cache operation and hence this lazy invalidation does not incur any additional reads to the cache. However, in rare situations, it may happen that all the cache sets are not accessed in an entire epoch. So, before transitioning to the new epoch, the cache sets that are still untouched during the entire epoch are read and the

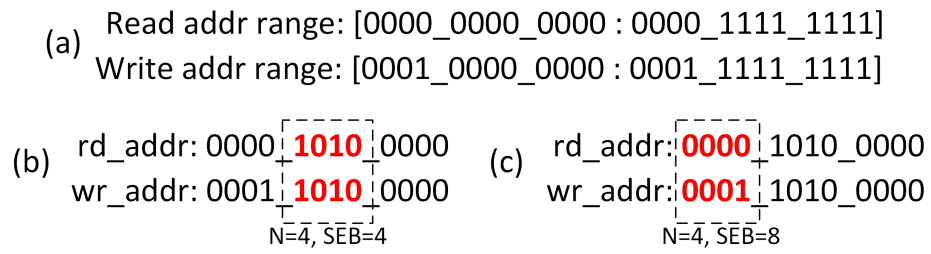


Figure 7.5: Adaptive band implementation. Our implementation of adaptive band formation increases or decreases the SEB by only one bit at a time. So, to move from a band formation shown in (b) to (c), our implementation will require four adaptive band transitions. The decision to shift up/down is based on whether the bits lower to the band position bits are different or not. If the lower bits are different, that means band position bits must be moved to lower bits so that the reads and writes will be in different bands, and vice versa.

cached addresses with bands corresponding to the current epoch are invalidated. Thus, the lazy invalidation mechanism may still need to do cache walk but on a very few targeted cache sets and not the entire cache. After this cache walk (which is rarely needed), the CU moves to the new epoch by responding with DoneAck.

7.1.3 Epoch Skipping Implementation

The baseline STC implementation imposes epochs for all bands in the address space, regardless of whether there are any pending writes to those bands. Epoch skipping considers this case and elides epochs without any pending writes. As described, STC blocks the writes not belonging to the current epoch. With the epoch skipping optimization, the blocked writes send an *EpochDemand* request to the EMU. The EMU keeps track of the epoch demands in an epoch request vector (ERV). The ERV is a bit vector with one bit per epoch. On receiving the epoch demand request, the entry corresponding to the requested epoch is set in the ERV. When the EMU wakes up, it increments the epoch counter till it finds an epoch with an outstanding demand in the ERV. This epoch then becomes the new epoch and EMU sends the PrepareEpochChange to initiate the transition to this new epoch after resetting

the ERV entry. If no entry is set in the ERV, EMU does not change the current epoch and goes back to sleep again.

On receiving an EpochDemand request, the EMU responds back with an EpochDemandAck notifying the requesting CU that its request is acknowledged. The CU issues only one EpochDemand per band, thus avoiding duplicate EpochDemand requests to the same epoch for that band. This EpochDemand request filtering is implemented with an epoch demand bit vector. Each CU has its own epoch demand vector and this vector has one entry for every epoch. A blocked store issues an EpochDemand request only when the corresponding entry in the epoch demand vector is empty. An entry is set after the first EpochDemand request thus preventing duplicate EpochDemand requests to the same epoch from a CU. The EpochDemand request vector is reset during the beginning of an epoch. This epoch filtering mechanism thus keep an upper bound on the number of outstanding EpochDemand requests from a CU and filters out duplicate requests, significantly reducing the number of such requests.

7.1.4 Adaptive Band Implementation

The adaptive band mechanism attempts to regroup addresses in a band in a way that avoids co-mingling of read-only addresses with read-write addresses using a simple dynamic search heuristic. Figure 7.5(a) shows an example where the first 256 addresses are allocated for read datasets and the next 256 addresses are allocated for write datasets. These types of contiguous allocations are common for GPU data structures with programmers optimizing the data layout for better reference coalescing. Figure 7.5(b) shows an example where both these read and write data sets share the same band with N=4 and SEB=4. With SEB=4, both the read_addr and write_addr are in $band_{10}$ and this read_addr in $epoch_{10}$ will be issued as a non-cacheable read (adhering to epoch rules) because of this sub-optimal band formation. We call this $epoch_{10}$ a conflicted epoch and its occurrence an epoch conflict. However, if the SEB is increased to 8, the read and write data addresses fall into different

bands, that is to $band_0$ and $band_1$ respectively as shown in Figure 7.5(c). With epoch skipping, an EpochDemand request to $band_0$ will not be issued and consequently, $band_0$ will never become the current epoch enabling read datasets to be cached throughout the execution.

Our implementation of adaptive bands dynamically increases or decreases the SEB to reduce the epoch conflicts. The epoch conflict detection requires both the conflicting read and write addresses for calculating the adaptive SEB. Towards this, we modified the EpochDemand request to send the entire write address and added an EpochConflict message to send the conflicting read address to EMU. EpochConflict message is generated when a CU has a blocked write to a band and then a subsequent read to the same band happens. To store the write address, each entry of the ERV is enhanced with a 32-bit address field. When an EpochConflict message arrives, the read address from that message is compared against the stored write address to identify the optimal SEB. The decision to shift up/down is based on whether the bits lower to the band position bits are different or not. If the lower bits are different, that means band position bits must be moved to lower bits so that the reads and writes will be in different bands, and vice versa. In our implementation, each adaptive band formation increases or decreases the SEB by only one bit. So, to move from a band formation shown in Figure 7.5(b) to Figure 7.5(c), our implementation will require four adaptive band transitions. But an aggressive implementation may shift multiple bits at a time and can reach the optimal band positioning much faster. Additionally, to reduce the number of EpochConflict messages from a CU, only the first conflicting read access generates an EpochConflict message. The EMU informs the CUs about the updated SEB by sending it along with the ChangeEpoch request. With the adaptive band formation, there may be overlap between the new bands and old bands. So, if there are pending memory operations during adaptive band transition, the memory operations may return in a different band than when they were issued. However, the four-way handshaking mechanism ensures that the CUs have no pending stores during epoch transition, thus avoiding such a situation for a

Table 7.1: State transition table

	Store	**Load**	**Load_U**	**Data**	**Data_E**	**WA**
I	I / PW++	I_V	I_I			PW−−
V	I / PW++	Hit	I_I			PW−−
I_V				V / LD	I / LD	PW−−
I_I				I / LD	I / LD	PW−−

store. The load may return in a different band, but the load data is not installed in the cache if it belongs to the current epoch (the current epoch check is done based on the new band definitions). Thus, overlap of bands during adaptive band transition does not violate STC rules.

Our adaptive band implementation only changes the start bit used for the address to band mapping. Thus, this new mapping ensures that all the addresses have a corresponding band and an epoch. The EMU, like before, round-robins through all the epochs thus ensuring writes to any address will eventually complete.

7.1.5 Multi-band Epoch

To implement the multi-epoch band optimization, current epoch register (CER) is enhanced to store multiple concurrent epochs. Also, the comparators that determine if a load/store is in the current epoch are enhanced to check for multiple epochs. The lazy invalidation mechanism discussed earlier is also augmented to check for multiple concurrent epochs. Other than the enhancements to these comparators and the CER, multi-epoch band implementation works similar to the adaptive band. The additional epochs to be granted are determined similar to the adaptive band optimization. However, instead of just selecting one epoch with pending epoch demand in a round-robin fashion, its adjacent epochs with pending epoch demands are also selected, and these epochs are granted together with the multi-band optimization.

7.1.6 Spatiotemporal Coherence and Cache States

The STC cache controller state transition table is given in Table 7.1. The epoch messages do not interfere with the cache states and hence are omitted from the table. In that table, *pending writes*, *load done*, and *write completion ack* are abbreviated as PW, LD, and WA, respectively. I is the invalid state, V is the valid state and I_V is the transient state when the cache is waiting on a load return to move to V from I. This transition table is similar to a GPU VI protocol except for the two events Load_U and Data_E, and one transient state I_I. A Load_U is a non-cacheable load issued when the current epoch matches the load's band and hence does not allow installation of the data in the cache. I_I is the transient state that tracks this pending Load_U and moves to I when the load returns making it non-cacheable. The Data_E is the Data response to a load whose band matches the current epoch and hence the data from that epoch is also not installed in the cache even if that load was issued as a cacheable load (Load) in one of the previous epochs (I_I to I).

Atomic operations are resolved in the shared L2 in GPUs [14]. In STC, atomics are treated as regular writes. Atomics that return a value generate a Data_E response along with a write completion ack (WA) wheres atomics with no return value only generates a WA. Hence, atomics do not have additional states in the cache transition table.

Protocol Complexity

STC adds one transient state I_I to the RC managed baseline. However, an invalidation-based hardware GPU VI protocol will add 4 to 9 transient states to the baseline [98] depending on the optimizations involved. Transient states increase the protocol verification complexity [35, 102]. Additionally, race transitions [102] that further adds to the verification complexity are also absent in STC. Because of these reasons, STC is easier to verify than any invalidation-based coherence protocol. STC is a rule-based system and adhering to the rules provides correctness. Hence, the correctness invariant is to check against the STC rules, which is straightforward.

7.1.7 Hardware Additions

The EMU has an epoch counter and a wakeup counter. The epoch skipping and adaptive band optimization add ERV and ETV to the EMU. The ETV is a 1-bit vector with one entry per epoch. ERV has 33-bits (32 bits for address, 1-bit to store epoch demand to that address) per entry and has one entry per epoch. The CU is augmented with a 32-bits per entry BSQ. All these structures can be implemented with logic cells and do not require SRAMs.

7.2 Correctness of the Implementation

We explain our implementation enforcing RC with the code snippet in Figure 7.6(a1,a2). Figure 7.6(a1) is the CPU implementation of the code and Figure 7.6(a2) is the GPU implementation of the same code with the GPU wavefront performing all three memory operations with one wavefront instruction. We will explain the implementation using CPU terminology but the working is exactly similar for the GPU as well. Addresses U, V, W and X belong to bands with the epochs U, W, W and X respectively. Figure 7.6(b) shows the BSQ of thread 0 (BSQ 0) and cache of thread 1 (Cache 1) after these threads executed the first three stores in some epoch Z. The transition from this epoch Z to epochs W->V->U->Z->X->Z are shown in Figure 7.6(c to h). Since thread 0 is only doing stores, nothing is cached in thread 0's cache and hence not shown in these figure. Similarly, thread 1 is only doing loads and hence its empty BSQ is not shown either. Since U, V and W do not have write permissions in epoch Z (Figure 7.6(b)), these stores are moved to thread0's BSQ. Also, the loads to U, V and W by thread 1 install these lines in the cache with the initial value 0 whereas the load acquire (LdAcq, line:4) installs X=0 in to the cache. Since X != 1, the load acquire condition is not satisfied and the load acquire will retry. The acquire/release ordering is implemented with the s_waitcnt instruction inserted by the finalizer [14] (section 7.1.1). This s_waitcnt stalls the pipeline till all the older memory operations are completed. Hence the StRel of thread 0 (line:4) is blocked from execution till all the previous stores are completed.

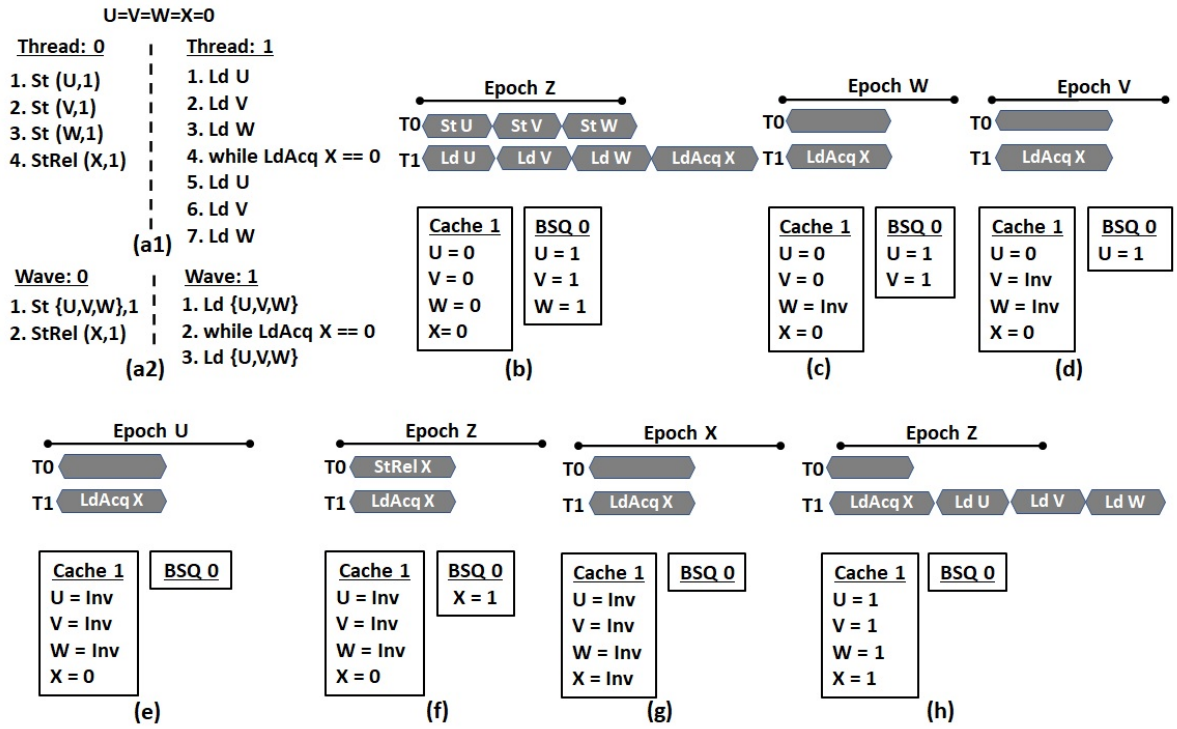


Figure 7.6: Demonstration of correctness with a working example. The pending stores from thread 0 (or wave 0) to U, V and W prevents the store release from executing {(a1),(a2),(b)}. This prevents the thread 1's (or wave 1) load acquire from satisfying its condition and executing the second set of loads to U, V and W {(a1),(a2),(b)}. Later, stores to W, V and U retires in their respective epochs {(c), (d) and (e)} after which the store release is executed (f). This store release to X retires in its epoch (g). Epoch X forces cache invalidation and thread 1's load acquire sees the latest update to X {(g) and (h)} that satisfies its condition. Thread 1 now executes the second set of loads and all of them sees the most up-to-date value (h), thus ensuring correct acquire-release synchronization operation. It should be noted that the order in which the stores to U, V and W retire is immaterial for the correct operation. The multi-band optimization that allows stores from multiple bands to simultaneously retire in an epoch is built on top of this property. As long as the epoch rules regarding the epoch synchronization and caching are followed, Spatiotemporal Coherence ensures correct write propagation.

Later, when Epoch W arrives, the store to W completes and that entry is removed from BSQ (Figure 7.6(c)). Additionally, the line W is invalidated from thread 1's private cache adhering to STC rules. Similarly, stores to addresses V and U are completed in epochs V and U respectively, and the cachelines V and U are invalidated from the private caches in these epochs (Figure 7.6(c,d)). Thus, by the end of epoch U, all stores are completed and the cachelines U, V and W are invalidated. Throughout these epoch changes, thread 1's LdAcq is reading X = 0 and thread 1 continues to spin at that instruction. The store completion acknowledgements are sent to the core after the stores are completed in their own respective epochs and the store release can now be executed by thread 1 as shown in Figure 7.6(f). However, the released store (X = 1) sits in the BSQ till epoch X arrives and is drained from the BSQ later in epoch X (Figure 7.6(g)) marking its completion. At the beginning of epoch X, the cacheline X is invalidated in thread 1's private cache and a subsequent read to address X will now see the updated value (Figure 7.6(h)). Only after the load acquire condition is satisfied, the loads to U, V and W (lines:5,6 and 7 of Figure 7.6(a1)) are executed by thread 1. These loads will also see the most up to date value from the shared L2 cache and the outcome will be (U,V,W) = (1,1,1) thus enforcing RC. It should be noted here that a load acquire did not invalidate private caches (as discussed in section 7.1.1) and still achieved the write propagation by following the STC's epoch rules.

7.2.1 Ordering of Writes

In the previous example, the stores are executed in the order U->V->W (Figure 7.6(a1)) but are completed in the order W->V->U (Figure 7.6(c,d,e)). Since the implementation was adhering to RC model, maintaining store->store ordering was not a requirement. The only requirement was to order stores U,V and W before the RC synchronization marker, that is the store release to X, and that requirement was enforced by the core by blocking the execution of the store release till the completion of older stores. A strong model implementation on top of the STC can enforce a stricter store->store ordering by blocking the issue of a younger store

till the older store completes. But our RC implementation did not have that requirement and hence allowed out of order store completion within RC model permitted synchronization marker boundaries.

7.2.2 Correctness with Multi-band Epoch Optimization

In the above example, bands W, V and U took three epochs to complete the stores. With multi-band optimization, epochs of these three bands can be granted simultaneously. That means epochs U, V and W can be granted simultaneously and all three writes from the BSQ can be drained in this multiband epoch. The BSQ may be drained and stores may be completed in a different order compared to the above example. But granting epochs W, V and U simultaneously still functions correctly and produces the same outcome (U,V,W) = (1,1,1).

7.3 Evaluation

Here we present simulation results for execution time and other secondary metrics for the following four versions of STC:

1. STC-NV: Naive STC implementation without any optimizations.
2. STC-ES: STC with epoch skipping optimization.
3. STC-AB: STC-ES with adaptive band optimization.
4. STC-MB: STC-AB with multi-band optimization. In our implementation, we allowed up to 4 simultaneous bands.

Additionally, we also compare STC against the temporal coherence proposed in the prior work [98]. We also evaluated a cacheless baseline (disabling L1 but keeping L2) but it performed so poorly for some benchmarks (up to 62.13% slower) we decided to instead baseline against a realistic current-generation design with RC-managed caches [11, 47, 49,

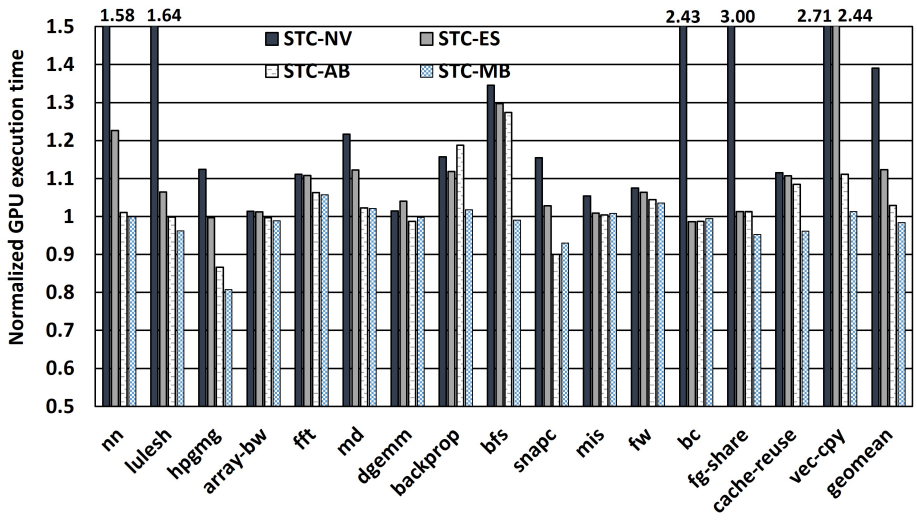


Figure 7.7: GPU execution time normalized to the baseline. STC with the multi-band optimization achieved 1.63% speedup over the baseline across all evaluated workloads. The benchmarks with increased cache hit rate (lulesh, hpgmg, snapc, bc, cache-reuse) because of data reuse across kernels observed a geomean speedup of 7.13%.

80]. The baseline L1 data cache is 64KB but we will compare against a 16KB L1 cache in the sensitivity analysis section 7.3.2.

The EMU in our STC implementation wakes up every 100 cycles and changes the epoch. We simulate STC with 16 epochs (4 bits to identify a band/epoch, that is N=4). The default SEB is 12. The SEB is modified dynamically by the adaptive band optimization. However, since 12 bits indicate the page boundary and 32 bits (4 GB) is the maximum addressable physical memory available in some GPUs, we decided to limit the SEB in the range of 12 to 32. STC specific simulation parameters can be found in the Table 7.2.

Table 7.2: Spatiotemporal Coherence specific simulation parameters

Number of Epochs	16 (N = 4-bits)
EMU Clock	2GHz
EMU Wakeup Cycles	100 GPU cycles
Default SEB	12
BSQ (and coalescing buffer)	256 entries

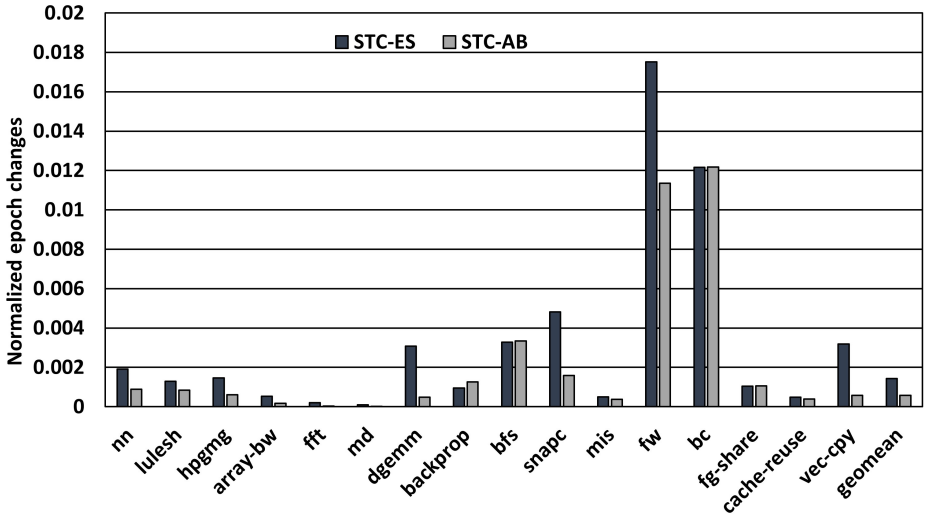


Figure 7.8: Decrease in the number of epoch transitions. STC-ES and STC-AB are compared against STC-NV. STC-NV implementation changed epochs at fixed time intervals even if the GPU was inactive resulting in a large number of unnecessary epoch transitions. The optimizations changed epochs only on demand - i.e. when a write is pending to an epoch - resulting in the nearly 3 orders of magnitude reduction.

7.3.1 Execution Time

Figure 7.7 shows the GPU execution time of benchmarks normalized to the baseline. It can be seen that the unoptimized STC (STC-NV) performed significantly worse than the baseline with the mean runtime increasing by as much as 39.04% across all evaluated benchmarks. However, the epoch skipping optimization alone regained most of the lost performance and STC-ES was only 12.29% slower than the baseline. STC-AB further brought down the performance penalty to 2.93%. However, the full optimization enabled STC-MB improved the performance by 1.63% compared to the baseline.

STC-NV blocks the writes and invalidates the caches during every epoch. Additionally, the average time to change an epoch was observed to be 36 cycles (including time for the four-way handshake) and, writes are blocked during this epoch transition as well. All of these contributed to the execution slow down. STC-ES filters out the epochs without any write requests and mitigates both these problems and thus improves performance. The reduction

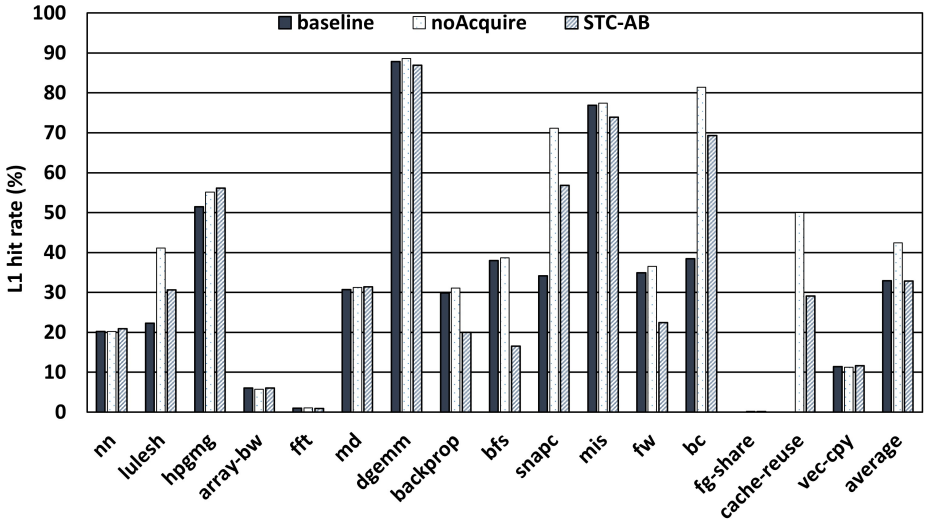


Figure 7.9: GPU L1 cache hit rate. On average, the adaptive band optimization was able to retain baseline hit rate even with epoch induced invalidations.

in the number of epoch transitions as compared to STC-NV is shown in Figure 7.8. Both STC-ES and STC-AB observed extreme reductions in the number of epoch transitions. With epoch skipping optimization, both STC-ES and STC-AB only transition to epochs that have writes pending to them and this eliminates unnecessary epoch transitions. Figure 7.8 also shows that the STC-AB optimization reduces epoch transitions by three orders of magnitude, which is also reflected in the execution time reduction shown in Figure 7.7.

Figure 7.9 shows the L1 hit rate of the baseline compared to the STC-AB. That figure also shows the hit rate after disabling the acquire operation. The benchmarks that show increased hit rate without acquire operation have read data reuse across kernel launches and can potentially benefit from STC's acquire-less operation. From that figure, lulesh, hpgmg, snapc, bc, and cache-reuse exhibit this behavior. For these benchmarks, it can be seen that STC-AB improved the hit rate compared to the baseline.

Figure 7.9 also shows that STC-AB restored the hit rate to baseline levels for most benchmarks. However, the performance (Figure 7.7) was still behind the baseline. This suggests that the cache hit rate has little impact on the performance degradation and the impact was

mainly caused by blocked writes as opposed to cache invalidations in an epoch. We mitigated this issue with the STC-MB optimization by allowing concurrent writes from multiple bands, reducing the number of blocked writes waiting on their epoch. This directly translated to performance as seen in Figure 7.7. With this final optimization applied, STC-MB achieved 1.63% speedup over the baseline across all evaluated workloads. More significantly, the benchmarks with increased cache hit rate (lulesh, hpgmg, snapc, bc, cache-reuse) because of data reuse across kernels observed a geomean speedup of 7.13%. Next, we delve deeper into the performance of a few benchmarks.

snapc,lulesh: The computation progresses in time steps in snapc, with multiple kernels (2992 kernels) launched to complete a time step. Similarly, lulesh launches 3380 kernels. However, many of these kernels have read data reuse. Since the baseline GPU employs software-managed cache coherence, the caches are invalidated at every kernel boundary (acquire operation at the beginning of a kernel) and the baseline cannot take advantage of this reuse. Thus STC achieves a higher cache hit rate and better performance for these benchmarks.

fg-share, hpgmg: fg-share uses atomic operations to synchronize the workgroups that are contending to enter into the critical section. Inside the critical section, each workgroup reads the updates from the previous workgroup and then modifies them. Since the shared data structure is updated in place, the read and write datasets are not separate. Consequently, the STC-AB could not identify a read-only dataset and performed similar to the STC-ES (Figure 7.7). The performance loss can be attributed to the write propagation delay of the atomic synchronization. This synchronization has to wait for its turn behind the shared data structure's epoch. But with STC-MB granting epochs for synchronization variable and shared data structure simultaneously, STC achieved 4.75% speedup against the baseline.

The read-modify-write access pattern of the shared data structure does not provide any caching benefit (Figure 7.9). However, the baseline allocates these loads in the caches, creat-

ing allocation/deallocation overhead (tag array lookup and associated port/bank conflicts, data allocation overhead including data array port/bank conflicts, eviction overhead) and thus delaying the load completion [58, 101]. STC-MB issues these loads as uncached, thus completely bypassing the cache (after all cache sets are accessed once for lazy invalidation mechanism, section 7.1.2). Thus loads are completed faster with STC-MB, resulting in speedup against the baseline.

hpgmg benefited from both read-data reuse across kernel launches as seen from its improved hit rate (Figure 7.9) and also from issuing 33.75% loads as uncached similar to fg-share.

vec-cpy: Each work-item in a vector copy benchmark copies one data element from a source to a destination buffer. This presents a pathological case for STC because the compute to memory ratio is low, and the GPU has less opportunity to overlap long-latency memory accesses with work. Since simultaneous writes from multiple work-items across the GPU result in writes to multiple epochs at all times, epoch skipping could not find many epochs to skip as observed from a large number of epoch changes by STC-ES in Figure 7.8. Consequently, STC-ES performed poorly (Figure 7.7). However, the adaptive band optimization could take advantage of the read-only source and write-only destination buffers, thus regaining most of the lost performance (within 11.1% of baseline). STC-MB further improved the performance to within 1.3% of the baseline even for this pathological benchmark.

It should be noted that STC's primary design objective is a coherence protocol with low hardware overhead and low coherence message overhead (Section 1.1) that also matches the performance of the baseline software-managed coherence protocol. From our evaluation, STC achieved this performance objective with 1.63% speedup over the baseline across all the evaluated benchmarks. We will show that it satisfies the low hardware and coherence message overhead requirements in Section 7.3.3.

7.3.2 Sensitivity Analysis

Sensitivity to Epoch Configurations and Cache Size: We experimented with 50, 100, 150, 300, and 450 cycle epochs, and the performance varied by 3%. The best performing 100 cycle epoch was used for our evaluations as mentioned in Table 6.1. The epoch duration primarily impacts the latency of the stores blocked by STC. Store completion delay can stall a wavefront if either (a) dependent reads are delayed because of pending older writes, or (b) wavefront is waiting on a release operation. Even stalling one wavefront will not impact performance because the GPU pipeline has the ability to switch execution to the next ready wavefront. Because of these GPU characteristics, store latency, and consequently epoch cycle duration had little impact on the GPU performance. Also, we ran experiments with 8, 16 and 32 epochs, and the performance variation was less than 2%. With 16kB private cache, STC outperformed the baseline by 1.97%.

Sensitivity to BSQ Size: The max occupancy of the 256-entry BSQ was found to be between 5 and 250 entries and the mean occupancy was between 0.5 and 102 entries for the evaluated benchmarks. The 16-banked BSQ is capable of draining up to 32 entries in 1 cycle (2 entries drained per bank per cycle, 8 cycles for draining all 256 entries) and most of the benchmarks (14 out of 16) have a mean occupancy of less than 32. Thus, these benchmarks were able to cycle through the BSQ in 1 cycle on average. Even for the benchmark with a mean occupancy of 102 entries (fft), the draining overhead was 4 cycles indicating that draining BSQ is not a bottleneck during epoch transitions. Additionally, only entries to the current epoch are issued to the cache. Thus the issue rate to cache is much less.

Figure 7.10 shows the performance of benchmarks for BSQ sizes ranging from 64 entries to the baseline 256 entries. The performance was relatively insensitive to BSQ size with the geomean across all benchmarks comparable to the 256 entry baseline. As mentioned earlier, 14 out of 16 benchmarks only had a mean BSQ occupancy of less than 32 entries and hence the benchmarks remained insensitive to the evaluated BSQ sizes. A BSQ size of less than 32 entries could have made a performance impact. However, the evaluated GPU has a 64

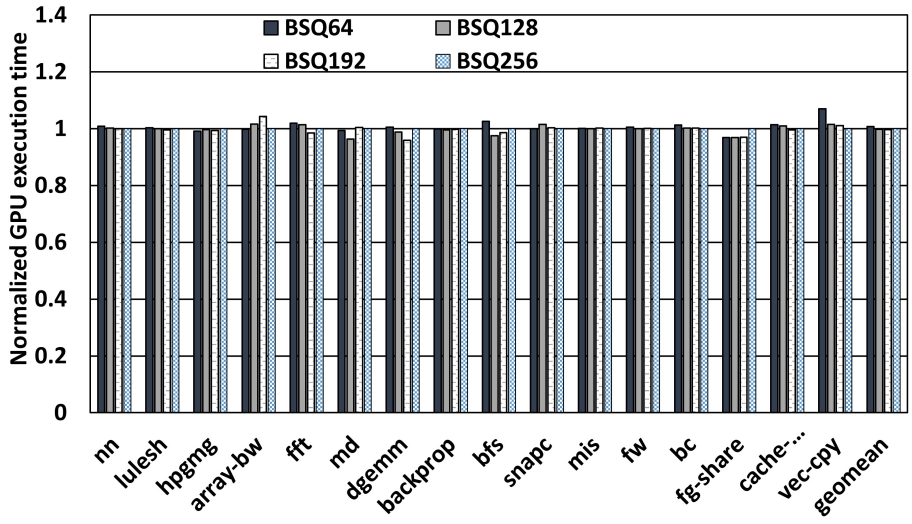


Figure 7.10: Sensitivity to blocked store queue size. Only two benchmarks had a mean BSQ occupancy of more than 32 entries. Hence the relative insensitivity to BSQ sizes from 64 to 256 entries. Since the GPU has a 64 thread wavefront, the BSQ must have atleast 64 entries to accommodate up to 64 uncoalesced stores originating from a single wavefront instruction. This prevented us from evaluating a BSQ size less than 64.

thread wavefront and hence the BSQ must have atleast 64 entries to accommodate up to 64 uncoalesced stores originating from a single wavefront instruction.

7.3.3 Bandwidth and Hardware Cost

Figure 7.11 shows the increase in bandwidth consumption due to epoch requests. STC-NV consumes significant bandwidth because it changes the epochs even when the GPU is idle (e.g., between two kernel launches). The figure also shows the bandwidth consumption of STC-ES and STC-AB. With STC-ES, epochs are changed only when the GPU demands an epoch and hence the epoch change is naturally stopped when the GPU is inactive resulting in the reduction of bandwidth. STC-AB further reduces the epoch transitions by changing the bands to epoch mapping. As such, the GPU bandwidth increased by only 0.43%, despite these optimizations sending additional messages. Hence, the impact of the optimized STC implementations on bandwidth can be considered a nonissue.

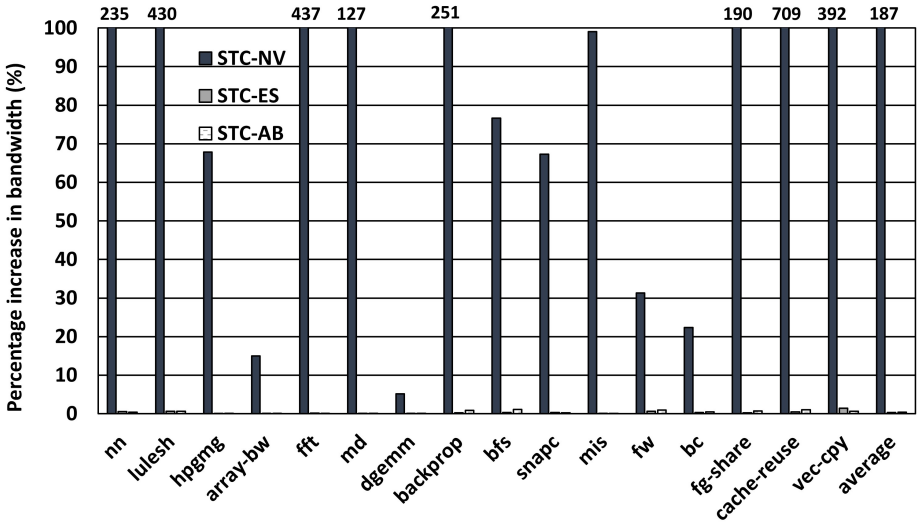


Figure 7.11: Bandwidth consumption of Spatiotemporal Coherence compared to the baseline. STC's bandwidth overhead was only 0.43% over the baseline and can be considered a non-issue.

The bandwidth cost of a purely hardware based implementation for a similar set of benchmarks (8 of the 16 benchmarks are common) is discussed in Section 9.4.2. Compared to the 17% increase in GPU bandwidth reported in that prior work because of the invalidation and eviction traffic, STC is far better with only 0.43% increase in bandwidth consumption over the baseline.

STC's low bandwidth consumption even for fine-grained data sharing benchmarks can be explained by a comparison of its operation to a traditional coherence protocol. Suppose few cores are performing a producer-consumer data sharing via a data structure. In a traditional protocol, the modifying cores send write requests to the directory, the directory then sends invalidations to the sharers and the owners forwards the latest data along with exclusive permissions to the requestors. This sequence is performed for every single address being modified in that data structure and most importantly repeats for every single producer-consumer exchange. With STC, an epoch or few epochs (if the entire data structure cannot be grouped into the same band) are granted to this data structure in response to epoch

request(s), and the producer-consumer exchange can continue without any further coherence messages. With multiband optimization, the epoch(s) to this data structure can be always granted and these cores can perform repeated producer-consumer exchanges without issuing any further coherence messages for all addresses in that data structure. This is the reason STC's bandwidth overhead was only 0.43% even with fine-grain data sharing benchmarks (fg-share, bc), and coarse-grain data sharing benchmarks (snapc, fw, mis).

Table 7.3: Hardware cost of Spatiotemporal Coherence implementation

EMU	4-bit epoch and 8-bit wakeup counters 16 entry ERV (33-bits per entry) 16 entry ETV (1-bit per entry)
BSQ	1 per CU, 256 entry, 32-bits per entry

Table 7.3 lists the hardware cost for implementing the optimized STC-AB protocol with 16 epochs and the EMU waking up every 100 cycles. It can be seen from the table that the hardware cost of implementing the EMU is negligible.

7.3.4 Scalability

Figure 7.12 shows the performance of benchmarks for 8, 16 and 32 CUs normalized to their respective baselines. The geomean1 reports the mean performance across all evaluated benchmarks whereas geomean2 reports the mean performance of STC friendly benchmarks with data reuse across kernels (lulesh, hpgmg, snapc, bc). The mean performance was observed to be relatively same across configurations with different number of CUs. However, benchmarks like nn, dgemm and vector-copy observed a performance slowdown due to the increase in the epoch synchronization overhead (increased from 36 GPU cycles with 8 CUs to 79 GPU cycles with 32 CUs at steady state after the read-only bands are separated by the adaptive band optimization). But the STC friendly benchmarks observed performance benefits across the range because they were able to exploit STC's caching benefits and nullify the negative impact of the higher epoch synchronization overhead. Thus, it can be concluded that STC scales well for workloads with read-data reuse across kernels. The two benchmarks

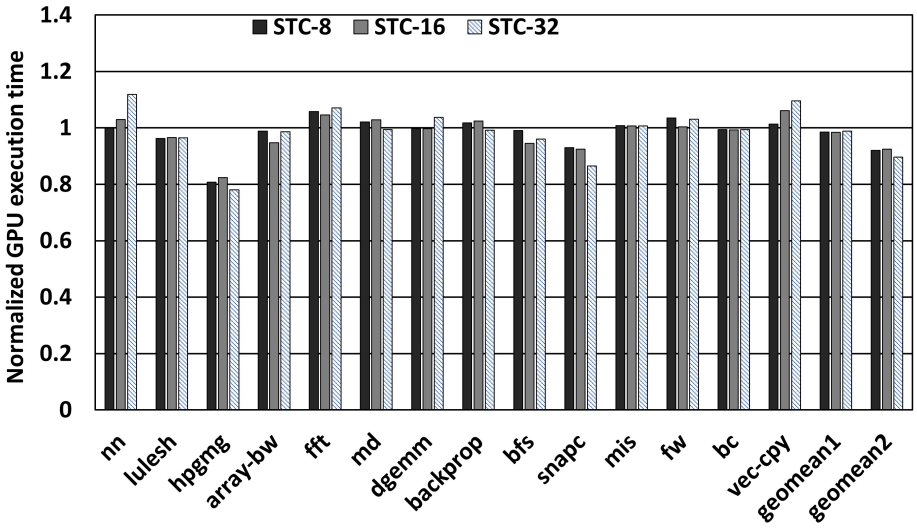


Figure 7.12: Scalability of Spatiotemporal Coherence. Scalability is impacted by the increase in epoch synchronization overhead with the number of CUs. However, the STC friendly benchmarks with data reuse across kernels (lulesh, hpgmg, snapc, bc) observed performance benefits across the range (geomean2) because they were able to exploit STC's caching benefits and nullify the negative impact of the higher epoch synchronization overhead.

fg-share and cache-reuse did not have enough workgroups to scale to 32 CUs with the same occupancy (number of wavefronts simultaneously active on a CU) as the baseline. Running GPU with a lower occupancy will hamper its ability to hide memory latency and hence these benchmarks are omitted from the results.

7.3.5 Comparison with Prior Work

We compared STC with the strong variant of the timestamp-based temporal coherence [98] (TCS) because it also provides write-atomicity similar to STC. Across all evaluated benchmarks, the TCS was slower than the STC by 3.46%. Although both STC and TCS avoid invalidation messages, TCS's exclusive modifier requirement delays a write till the global lease on a block expires. This makes the TCS slower than STC, especially for fine-grain data sharing operations.

While performance advantage is not significant, STC has storage overhead and hardware complexity advantages. TCS requires every cache block to store additional 32-bit lease information. Additionally, to store the lease information of evicted blocks from the shared L2, the TCS employs MSHRs (we used 128 entry MSHRs per L2 bank). Thus, for the simulated system with 8192 (8 CUs and 1024 cache blocks per CU) cache blocks in both GPU L1 and L2, and with 512 MSHR entries across four L2 banks, the TCS requires 66kB of storage to store lease information whereas STC requires only 8kB additional storage for its BSQs (8 BSQs and 256 32-bit entries per BSQ). The MSHRs not only add storage overhead but these associatively looked-up structures are difficult and costly to implement beyond a few entries [80, 84]. STC does not require metadata storage and associative structures for its implementation. Additionally, TCS sends 32-bit timestamp information with every read request whereas STC sends additional epoch requests for enforcing coherence. A comparison of the bandwidth requirement shows that TCS increased the memory traffic by 3.57% against STC's 0.43% across all evaluated benchmarks.

7.4 Chapter Summary

We implemented, demonstrated the correctness with a working example, and evaluated STC in this chapter (Sections 7.1, 7.2 and 7.3). Our evaluation shows that STC improves the performance of workloads with read data reuse across kernels by 7.13%, and provides performance comparable to the software-managed baseline coherence implementation for the rest of the evaluated workloads. STC also performs better than prior work. It achieves this with very low hardware overhead and virtually no coherence message overhead while also providing write atomicity. Thus, it satisfies all the criteria for a low overhead GPU coherence implementation that can also support strong models.

We implement and evaluate Entangled Coherence in the next chapter.

Chapter 8

Entangled Coherence for Heterogeneous SoCs

In this chapter, we first discuss the implementation of Entangled Coherence followed by a section discussing the correctness of the implementation (Sections 8.1 and 8.2). We conclude this chapter with an evaluation of our implementation (Section 8.3) and a chapter summary (Section 8.4).

8.1 Implementation

We implement Entangled Coherence on a heterogeneous SoC system emulated by a GPU with processing engines of the SoC emulated by compute units (CUs) as discussed in section 6.2.3. Each pipeline stage of the software pipeline generator application (Section 6.3) runs on a CU. Entangled Coherence works by grouping CUs that need to be synchronized together to form an entangled group. For creating entangled groups, the epoch management unit (EMU) must know the sharing pattern between the CUs. Towards this, the CUs first form bands by combining addresses with the same access type to the same band, thus forming read, write and read-write bands. Once the bands are formed, the CUs advertise them to the epoch management unit (EMU). Once EMU has the bands along with their access types

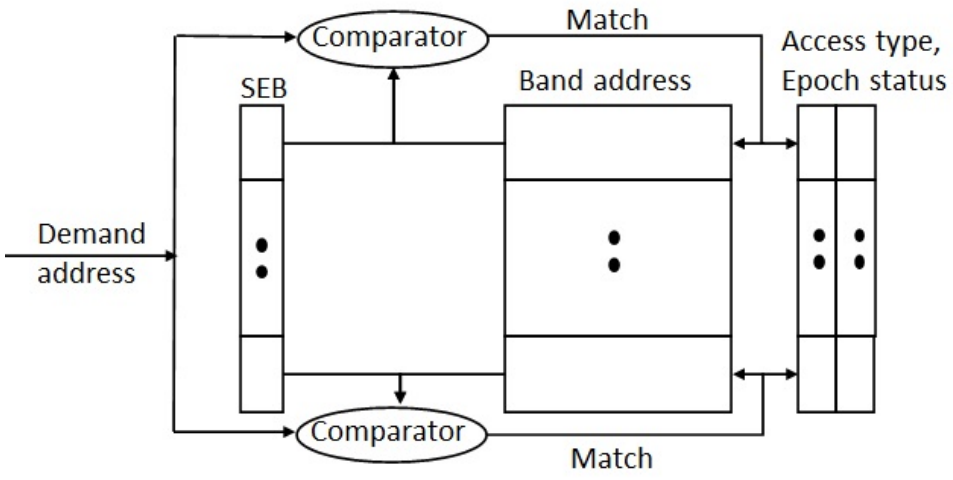


Figure 8.1: Band map. Each CU is augmented with a band map that stores the band information such as band address, start epoch bit (SEB) and access type to that band. The band map also has a single bit per entry to store the current epoch status - i.e. granted or not granted - of that band.

from all CUs, it can form entangled groups and establish entanglement across various CUs. Before entanglement is established, all epochs are granted by the EMU and hence at the beginning of execution, the GPU acts like a cacheless system.

8.1.1 Band Formation

For forming bands, the CUs must first know their read and write address ranges. Figure 8.1 shows the band map added to CUs for storing their band information and access types. The band map is a fully associative structure with each entry having a start epoch bit (SEB), the band address and the access type. The start epoch bit is used to map the band address to its corresponding epoch. The default SEB is set to 12 matching the typical 4KB page size. An additional epoch status bit in each entry stores whether that band's epoch is currently granted.

Before entanglement is established, all epochs are granted by the EMU and the GPU acts like a cacheless system. Thus, in this initial phase, no stores are blocked and all loads

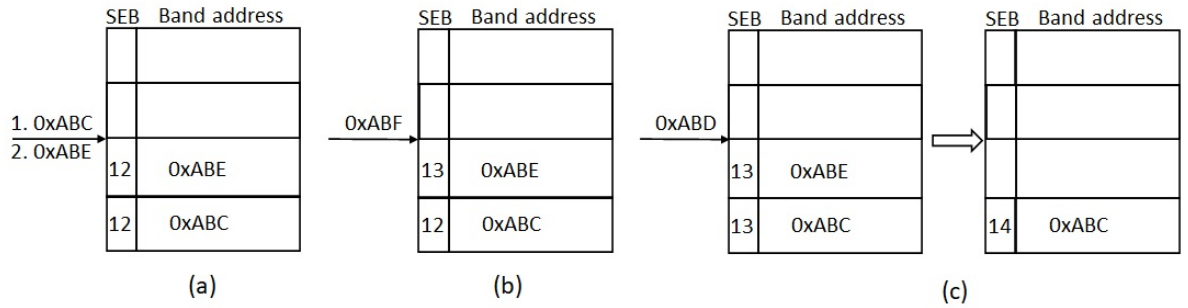


Figure 8.2: Demonstration of band map compaction algorithm. The bands with physical frame numbers 0xABC and 0xABE are not adjacent bands and hence they are allocated separate entries (a). The adjacent bands 0xABF and 0xABE are compacted to a single entry as shown in (b). The compacted entry's SEB is also incremented by 1. Later, when another band 0xABD arrives, it is merged with entry 0xABC and its SEB is also incremented by 1. Now, the entries with similar SEBs are analyzed to see if they can be merged together and entries 0xABE and 0xABC are merged to a single entry after incrementing their SEB to 14 (c).

are issued as uncached. When a memory access is issued by a CU, the epoch of its demand address is computed based on the SEB in the band map and the epoch address is compared against the existing entries to find a match. If there is a match, the stored access type is compared against the access type of the demand access and the access type is appropriately updated in the band map entry. For example, if the stored access type mismatches with the demand access's access type, the stored access type is updated to read-write. Otherwise, the stored access type is not updated. If no match for this demand address is found in the band map, a new entry is added to the band map for this demand address. Its SEB is set to 12 (default SEB) and the access type is updated.

The scheme described above will result in one entry per 4KB page in the band map. Such an implementation is not practically feasible. Moreover, our objective is to group contiguous pages with the same access type to a band so that the number of bands and correspondingly the number of epochs that has to be granted are a minimum. Towards this, we implemented a band map compaction algorithm.

Figure 8.2 demonstrates this compaction algorithm in detail. Figure 8.2(a) shows a

scenario where the first demand access with a physical frame number (PFN) 0xABC is issued by the CU and is inserted to the band map. Its SEB is also set to the default SEB 12. 0xABC is the PFN, the actual demand address is 0xABCXXX but the page offset bits are discarded while storing it in the band map because for band to epoch mapping, our implementation's lowest SEB is 12. Later, when a new demand access with PFN 0xABE arrives, we compare entry 0xABC for a match with SEB 12 and also with SEB+1 (=13) to see if the new PFN is to an adjacent frame to the already stored band. Since PFN 0xABE is not an adjacent frame of 0xABC, it is added as a new entry. However, later when a new access with PFN 0xABF arrives (Figure 8.2(b)), since this frame is an adjacent frame of 0xABE, the comparison with SEB+1 for the 0xABE entry will produce a match and hence this new entry is combined with the existing entry after updating the SEB to SEB+1. This is shown in Figure 8.2(b) where the 0xABE's SEB is incremented by 1. Whenever the SEB is incremented, an associative search on all entries with the updated SEB is performed to see if any other entries can be compacted. This is shown in (Figure 8.2(c). A new demand access with PFN 0xABD will find a match with entry 0xABC and that entry's SEB is incremented by 1. After incrementing the SEB, an associative search on the existing entries for matching SEBs find a match in entries 0xABE and 0xABC. Since they are to adjacent bands, they are compacted to a single entry and the SEB is incremented again. Since SEB is incremented, this process is again repeated till all adjacent bands are compacted.

We only compact adjacent bands with matching access types. For example, we do not compact the bands 0xABC and 0xABE in the previous example (Figure 8.2(c)) if they had different access types. This ensures adjacently allocated read and write data structures are not incorrectly grouped to a single read-write band. Synchronization variables typically see both read and write accesses to them and their page may be incorrectly grouped together with adjacent read-write pages (example, private/scratch region). To prevent this from happening, we allocate synchronization variables with guard pages. These guard pages are dummy pages without any accesses to them from any CUs and hence prevent the synchronization pages

from being combined with other bands. Thus, with the compaction mechanism discussed above, the CUs are able to group contiguous addresses with same access type to a band with its own band address, SEB and access type. It should be noted that after compaction, the number of band map entries are reduced to just a few. For example, for a pipeline stage with one producer and one consumer discussed in Figure 6.3, the band map will have just 6 entries (two for double buffered read frames, two for double buffered write frames, one for the scratch region and one for the synchronization variable *myCurrentFrame*).

While the band formation discussed above works well for most applications that we are targeting, there can be situations in which there are too many bands for the band map structure to track. This can happen with extremely fragmented memory allocation. In such situations, that is if the band formation discussed above fails for any reason, Entangled Coherence falls back to Spatiotemporal Coherence for providing coherence with the band formation now taken over by the adaptive band optimization discussed in Section 4.1.2. In another degenerate case, all the bands may be mapped to a single band and then Entangled Coherence always grants the epoch corresponding to that band forcing the system to behave like a cacheless system. Thus, correctness will be maintained in these degenerate scenarios but performance may suffer.

8.1.2 Establishing Entanglement

Once the bands are formed by the CUs and stored in their band map, the CUs send the band information of each band map entry to the EMU via EpochInfo messages. These EpochInfo messages carry the epoch address and the access type. An epoch address is derived from the band address using its SEB. For the rest of this discussion, an epoch address means a band address, SEB pair and comparing epoch addresses means comparing addresses [32:SEB] (we are simulating a GPU with a 32-bit address space). The objective of sending EpochInfo messages is to advertise the epochs and their access types to the EMU so that EMU can establish the sharing pattern among the CUs. The EMU on receiving this information adds

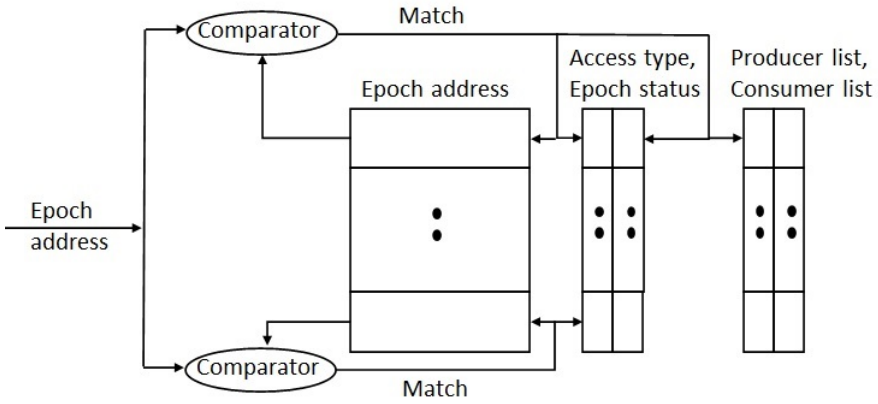


Figure 8.3: Entanglement map. Epoch management unit is augmented with an entanglement map that stores the epoch address, its access type and the producers and consumers of the band associated with that epoch.

it to its entanglement map.

The entanglement map is shown in Figure 8.3. It is similar to the band map of CUs but has two additional fields per entry: (a) producer list, and (b) consumer list. These lists stores the producer CUs and consumer CUs of each epoch. When an EpochInfo message arrives at the EMU, EMU compares its epoch address to find a match with any existing entry. If there is no match, the epoch address and its access type is added to a new entry. If there is a match and the access types mismatch, the access type is changed to read-write. Otherwise, no change is made. In both the matching and mismatching scenarios, the sender CU is added to the consumer list or producer list or both depending on whether the access type is read, write or read-write respectively.

The entanglement map is updated until all the EpochInfo messages from all the CUs are received by the EMU. Once all epochInfo massages are received, the EMU traverses the entries in the entanglement map to grant private epochs and synchronization epochs (epochs of synchronization variables) first. If an entry has only one producer and only one consumer and if they are the same, that is an epoch to a private band and they are immediately granted. The epoch granting mechanism is same as the 4-way handshake epoch

synchronization mechanism of spatiotemporal coherence (STC) discussed in section 7.1.2. If an entry has all the CUs in the producer and consumer list, that is a synchronization epoch and that is also granted immediately. The private epochs are granted with a private bit set in the epoch messages so that the CUs are notified about it. The CUs sets the epoch granted field of the band map Figure 8.1 for their granted epochs and additionally mark the access type to *private* for the private epochs. These epoch transitions complete after receiving the DoneAcks from all CUs similar to the STC implementation (section 7.1.2).

Once all the DoneAcks are received, the EMU has established entanglement among the CUs that coherently share data. Since synchronization epoch is granted to all CUs, the entangled group for the synchronization band consists of all the CUs. However, it should be noted that each CU now has been granted different private epochs. This is different from STC where all epochs are granted to all CUs. The CUs can issue writes only to the epochs granted to them. However, the CUs are not yet granted the epochs for coherently shared bands. Granting the epochs for coherently shared bands are discussed in the next section.

8.1.3 Operation under Entanglement

Once entanglement is established, no more updates to the band map or entanglement map are permitted. Each demand access now consults the band map of its CU to see if an epoch corresponding to its band has already been granted and also to determine if the access is to a private band/epoch. These demand accesses follows the epoch's caching rules (Section 3.1) and a store is issued only if an epoch is granted and loads to a granted epoch are issued as uncacheable. Additionally, the same lazy invalidation mechanism discussed in Section 7.1.2 invalidates the already cached entries belonging to the granted epoch. However, although private epochs are always granted, the loads and stores to a private epoch are issued as cacheable which forces even the stores to be allocated in the L1 cache. Consequently, a private store completion only incurs L1 access latency whereas a regular write-through store incurs L2 access latency.

For the non-private stores that do not have an epoch, they are moved to the blocked store queue (BSQ, refer section 7.1.2) and an EpochDemand request is sent to the EMU similar to STC. On receiving this request, the EMU immediately grants the epoch following the 4-way epoch synchronization mechanism but only to the producers and the consumers of that epoch listed in entanglementMap. Thus the entangled producer and consumer CUs only take part in this epoch synchronization making the operation localized to few CUs. It should be noted that our use case, that is the software pipeline application can have only one producer CU to an epoch because the write frames are written by only one pipeline stage (Section 6.3). Hence, if an epoch change that is not demanded by a CU is initiated on it, that CU is a consumer CU and can immediately ack the epoch synchronization requests without waiting for the pending stores to complete because those pending stores are anyway not to the epoch being transitioned into. This also speeds up the epoch synchronization. Additionally, unlike STC, the EMU grants epochs immediately rather than delaying granting epochs till it wakes up at fixed intervals of time. This also enables faster write completion than STC.

Other than the private epochs and synchronization epochs that are always granted, the EMU grants only one other epoch that is requested by the EpochDemand discussed above. Since our benchmark uses double buffering, a CU toggles between two write buffers as discussed in Section 6.3.3 and consequently the granted epochs toggle between the two epochs corresponding to these write buffers. Hence, only one EpochDemand request per frame is sent by a CU in our implementation emulating a pipeline stage (Section 6.3). This toggling of granted epochs also has caching benefits. For example, when a CU is writing to the current write buffer, its consumers are likely reading the previous frame from the second buffer. So, when a CU is only granted one epoch to the coherently shared data, it is naturally granted for the current buffer (and not for the second buffer), and hence consumer CUs can cache the second buffer because it is excluded from the epoch-induced invalidations.

8.1.4 Quasi-entanglement and Re-establishing Entanglement

The operation of Entangled Coherence under entanglement is remarkably similar to the STC's operation except STC grants epochs to all CUs. Entangled Coherence can ensure correctness even after granting epochs to only members of a band's entangled group because all other CUs are still considered to be quasi-entangled for that band. That means a CU is a quasi-entangled member of all the bands that are not in its band map. Consequently, if a CU did not find a matching band in its band map for a demand access, it sends an EpochInfo message to the EMU. An EpochInfo after the establishment of entanglement means the access pattern has changed and EMU responds by sending a reset message to all the CUs and resets its entanglement map. Once the reset message is received by CUs, they reset their band map after which the process of forming the bands and establishing entanglement starts again. In this way, Entangled Coherence can keep epoch synchronization local to few CUs via entangled groups and maintain correctness as long as the access pattern established during entanglement is not violated by any CU.

8.2 Correctness of the Implementation

We demonstrate the correctness of our implementation with the working example in Figure 8.4. The Figure 8.4 (a) shows a simple three stage pipeline. Each pipeline stage executes the pseudocode discussed in section 6.3.3. The pipeline is double buffered with CU0 writing to bands A0 and B0, and CU1 writing to bands A1 and B1. The bands A0 and B0 are consumed by CU1 whereas bands A1 and B1 are consumed by CU2. Consequently, CU0 and CU1 form entangled group EG_{01} via bands A0 and B0 whereas CU1 and CU2 forms another entangled group EG_{12} via bands A1 and B1.

Figure 8.4 (b) shows the timeline of band accesses by various CUs along with the epoch changes in each entangled group. The time difference between writes while switching frames (indicated by black arrow) denotes the epoch synchronization overhead. The band accesses

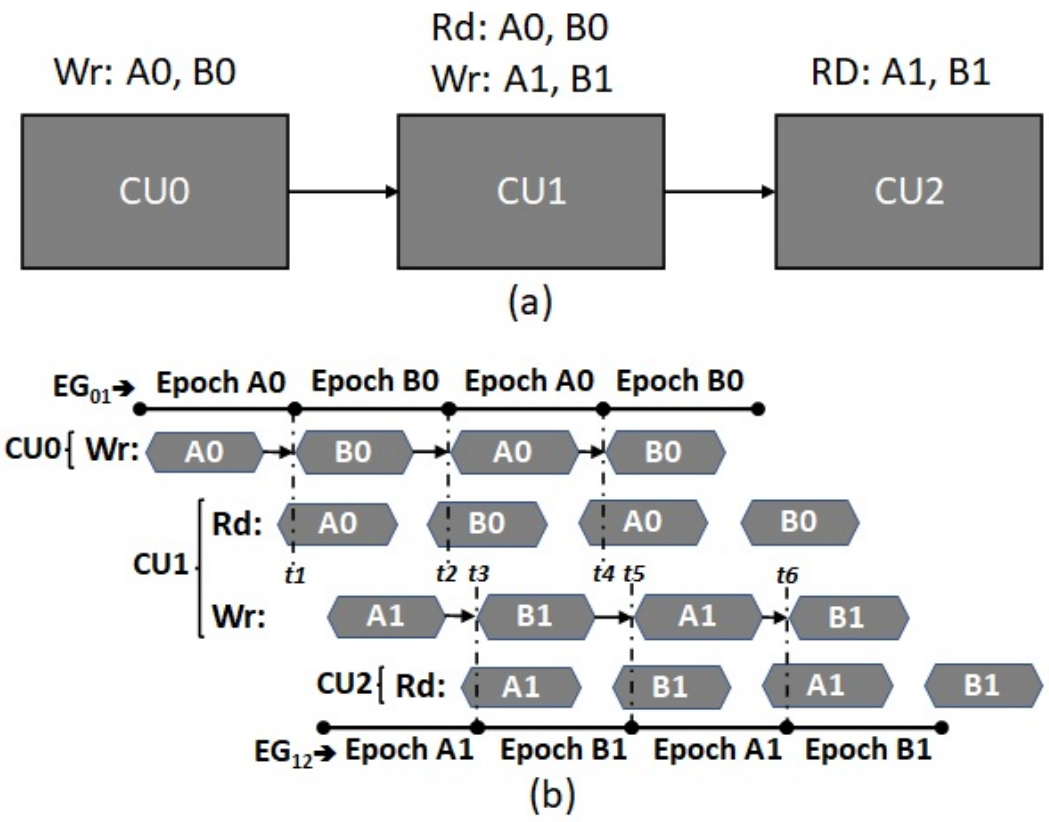


Figure 8.4: Operation after entanglement. (a) shows a three stage pipeline with CU0 and CU1 sharing the data via bands A0 and B0, and CU1 and CU2 sharing data via bands A1 and B1. The timeline of epoch transitions and band accesses are shown in (b). The epoch management unit synchronizes the CUs in the same entangled group while granting epochs and that ensures the correct write propagation as long as the CUs do not access bands that are not advertised to the epoch management unit during the entanglement establishment process.

by synchronization variables are not shown in this figure because since private epochs are always granted, they are always issued as uncached and cache coherence is not needed for those variables. STC's epoch rules ensuring correctness is discussed in detail in section 7.2. So, for ensuring the correctness of our Entangled Coherence implementation, we only need to ensure: (a) CUs within entangled groups follow STC's epoch rules, (b) the entangled CUs only access the bands associated with their entangled group, and (c) EMU correctly synchronizes the CUs in an entangled group. The Entangled Coherence implementation ensures (a) as discussed in section 8.1.3. Quasi-entangled state ensures (b) as discussed in section 8.1.4. We will demonstrate (c) with the help of Figure 8.4.

From Figure 8.4 (b), EG_{01} toggles epochs between epochs A0 and B0 at t1, t2 and t4. EG_{12} toggles its epochs at t3, t5 and t6. The read-after-write dependency on the bands is enforced by the programmer inserted synchronization operation and is discussed with the help of a pseudocode in section 6.3.3. Since CU0 writes to band A0 only after it is granted epoch A0, and since CU1 is also entangled with CU0 via the same band, the EMU synchronizes the band A0's epochs across both CU0 and CU1. Thus, both CUs are in the same epoch A0 till time *t1* as shown in Figure 8.4 (b). Hence, all the cached data to band A0 in CU1's private cache is invalidated in that epoch and CU1 reads the most up-to-date value of band A0. The updates to band B0 is also made visible to CU1 by the same process. For example, CUs 0 and 1 are in epoch B0 from time *t1* to *t2* as shown in Figure 8.4 (b). The write to epoch B0 only happens during this epoch and all stale data of band B0 is invalidated in this epoch from CU1's private cache thus ensuring correct write propagation. Similarly, CU1 and CU2 are members of EG_{12} and EMU synchronizes epochs to bands A1 and B1 across them. Thus, updates to bands A1 and B1 by CU1 are also correctly made visible to CU2 by following the epoch's caching rules.

8.3 Evaluation

In this section, we evaluate Entangled Coherence (EntC) using the two benchmarks *linear* and *autonomous* discussed in section 6.3.3. Both the benchmarks are generated by the software pipeline generator 6.3.3. The *linear* is a 5-stage linear pipeline with each stage feeding data to the next stage. Thus each stage has one producer and one consumer. This simple benchmark can be used for understanding the characteristics of EntC. The *autonomous* is the autonomous driving pipeline discussed in section 6.3.3 and shown in Figure 6.2. This represents a real-world use case. The linear transfers data with a 128KB frame whereas autonomous transfers data with a 1MB frame.

We simulated 128 frames of each benchmark. The benchmarks exhibit a fixed access pattern that repeats for every frame. The EntC establishes entanglement by 6-7 frames for both the applications after which the epoch activity also follows a fixed pattern that repeats for every frame. So, by the 8th frame, the coherence activity has reached a steady-state and we run them for an additional 120 frames so that the effects of the initial pre-entanglement phase is averaged out.

We evaluated 5 variants of each benchmark. These variants are generated by varying the parameters of the read, reuse, scratch, and write phases of each pipeline stage (section 6.3.3).

1. Stream: A variant with no reuse phase and no accesses to private/scratch area. This variant is generated by skipping the reuse and scratch phases of the pipeline stages (section 6.3.3). The read frames and the write frames are just accessed once making them streaming accesses.

2. Read-reuse: Same as stream variant but with the reuse phase added to the pipeline. The reuse phase issues accesses to the read frame. The accesses are done in such a way that all of them will be L1 cache hits. The read phase and write phase are still streaming. The reuse phase accesses the same amount of data as the read phase. That

means the maximum possible load hit rate is going to be 50% because 50% of the loads are streaming (from the read phase) and the remaining 50% will hit in the caches (from the reuse phase).

3. Scratch-reuse: Stream variant but with scratch accesses. The amount of accesses to the scratch space is kept the same as read and write frame sizes, but the accesses are made is such a way that they will generate cache hits provided the scratch/private writes are cached. It should be noted that the scratch/private accesses are typically a write followed by a read from the same processor. Hence, caching the write is essential for generating a load hit for the succeeding read.

4. Scratch-noreuse: Same as scratch-reuse but the access patterns are such that they will not generate cache hits. The scratch-reuse scenario is created by generating 128KB and 1MB writes (respectively from the linear and autonomous applications) to the scratch area in a burst so that the 128KB and 1MB reads that follow will miss in the 16KB L1 cache (section 6.2.3) although the reads are to the same scratch space.

5. Both-reuse: A mix of read-reuse and scratch-reuse variants.

The stream variant does not give advantage to either the baseline release-consistency managed caches or to the EntC. The read-reuse variant is supposed to perform poorly with EntC because of EntC's epoch induced invalidations reducing the effective load hit rate. Moreover, this variant does not have any scratch accesses either. So EntC cannot benefit from caching the scratch accesses by grouping them to a private epoch/band either. Hence the read-reuse variant represents the degenerate scenario for EntC. However, the scratch-reuse variant represents the best scenario for EntC with scratch accesses that are also guaranteed to be cache hits. The scratch-noreuse variant is included to evaluate the minimum benefit from EntC with scratch accesses. The both-reuse variant combines the best and worst scenarios (scratch-reuse and read-reuse respectively) for EntC to evaluate its performance in a mixed scenario.

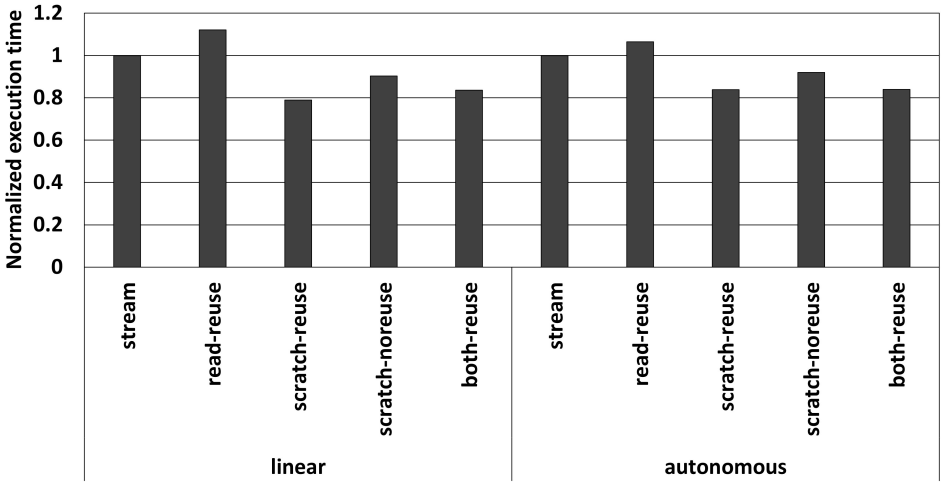


Figure 8.5: Performance of Entangled Coherence normalized to baseline. Entangled Coherence performed on par or better than the baseline for all variants but read-reuse. Read-reuse presents a degenerate scenario for Entangled Coherence in which the epoch's caching rules prevent a pipeline from caching data during its reuse phase.

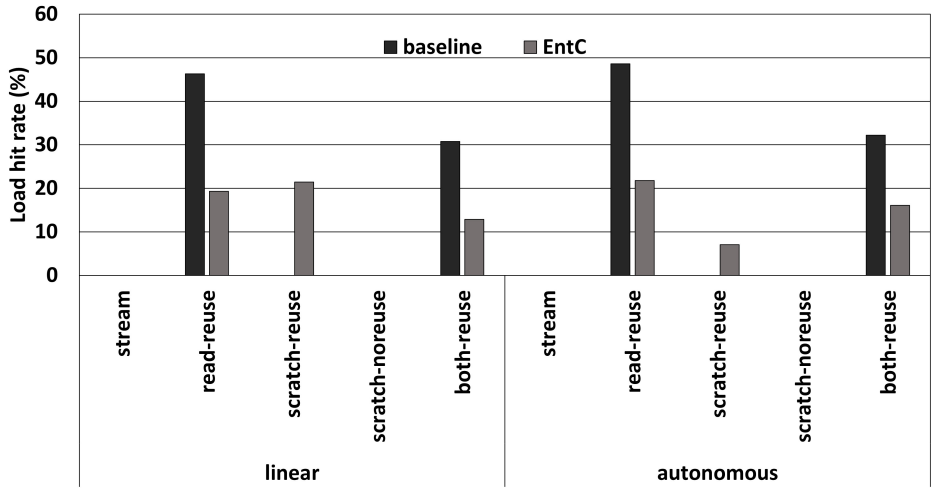


Figure 8.6: Load hit rate comparison. Baseline observed a higher load hit rate for variants with reuse phases. Entangled Coherence was not able to provide caching benefits to the reused data because of epoch's caching rules.

8.3.1 Performance Evaluation

Figure 8.5 compares the execution time of Entangled Coherence against the baseline. EntC performed better than the baseline for all the variants except stream and read-reuse for both the benchmarks. As discussed, stream does not have any data reuse or accesses to scratch region. It acts like a streaming benchmark, and both the baseline and Entangled Coherence behaved like a cacheless system for the shared read-write data sets thus resulting in similar execution time. Although Entangled Coherence has the additional overhead of requesting and granting epochs, it nullified this negative effect by issuing a fraction of loads as uncacheable (because of epoch rules) thus avoiding the allocation/deallocation overhead (tag array lookup and associated port/bank conflicts, data allocation overhead including data array port/bank conflicts, eviction overhead) that anyway could not have benefited from caching [58, 101]. This same phenomenon is observed for fg-share benchmark while evaluating STC and we have discussed this in detail in section 7.3.1. Thus EntC performed on par with the baseline for the stream variant.

However, issuing the loads as uncacheable resulted in performance loss with the read-reuse variant. Figure 8.6 shows the load hitrate of baseline and EntC for different variants of the benchmarks. The baseline observed a load hitrate of 46.29% whereas EntC could achieve a hitrate of only 19.29%. This resulted in baseline achieving significantly faster load completion during the reuse phase of the pipeline that resulted in the speedup.

The reason why EntC issues uncacheable accesses is demonstrated in Figure 8.7. The figure shows two CUs CU0 and CU1 sharing data via two bands A0 and B0. The write phase of CU0 and the read and reuse phases of CU1 are shown in that figure. Both CUs are in the same EG_{01} and the EG toggles between epochs A0 and B0 in response to epoch demand requests from CU0. After epoch A0 is granted to the EG_{01}, the next epoch demand to B0 will be initiated only when CU0 starts writing to B0 frame. However, there is a time window between the completion of writes to A0 and the starting of writes to B0. This time window lies between t1 and t2 in the Figure 8.7. During this time, CU0 will be synchronizing frames

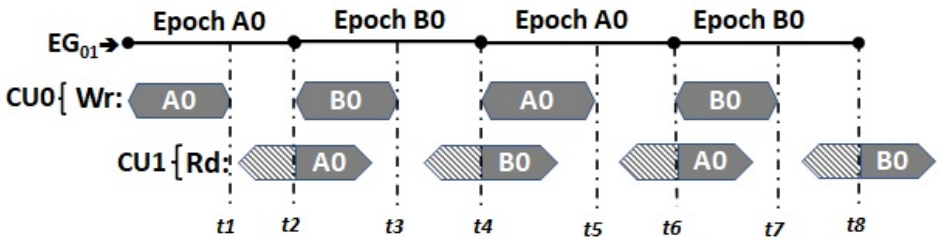


Figure 8.7: Uncacheable accesses with Entangled Coherence. Although no writes are happening in windows t1-t2, t3-t4, t5-t6 and t7-t8, the granted epochs still dictate the caching rules of the entangled group forcing all the CUs in it to issue loads as uncacheable. Since this window overlaps with the read/reuse phases of the consumer CU, the reuse phase's loads are forced to be issued as uncacheable thus lowering the load hit rate with Entangled Coherence.

with its producers and consumers, and executing its own read, reuse and scratch phases as discussed in section 6.3.3. However, the read and reuse phases of CU1 also falls in the same window (fully or partially depending on the length of the phases) as demonstrated in that figure. Since CU1 is also a member of EG_{01} and EG_{01} has the epoch A0 granted during this entire window, the accesses from the read and reuse phases are issued as uncacheable by CU1 thus lowering the load hit rate. This window of caching inefficiency repeats for every frame as shown by windows t3 and t4 (for frame B0), t5 and t6 (for frame A0) and so on in Figure 8.7 consequently leading to performance loss with the EntC..

This issue can be solved by an optimization that relinquishes the epochs after it has completed writing to its corresponding band. We leave this exploration to future work.

It can also be seen from Figure 8.7 that the load hit rate was lower for EntC for the both-reuse variant as well but it performed better than the baseline. For both-reuse variant, there are scratch accesses and EntC grants private epochs to these accesses (section 8.1.3) enabling both reads and writes to be cached in the private cache. Consequently, the private writes from the scratch accesses can be installed in the cache and a write completion ack sent back to the pipeline from the L1 cache incurring only L1 latency. Contrast this to the baseline where the pipeline has to wait for a write-through completion ack from the shared

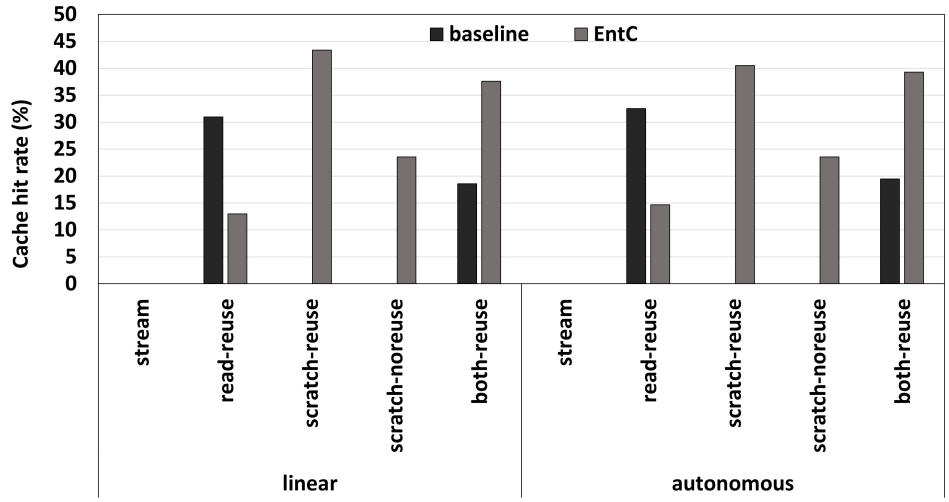


Figure 8.8: Cache hit rate comparison. Cache hit rate of variants with scratch phase improved with entangle coherence because of its ability to identify and group scratch accesses to a private band and provide caching benefits to them by granting them private epochs.

L2 incurring L2 latency. The effectiveness of this implementation can be understood from the cache hit rate shown in Figure 8.8. Here, a private store from EntC is categorized as a cache hit because it incur only L1 access latency as discussed above. As can be seen from that figure, other than the read-reuse and stream variants without scratch accesses, EntC observed significant cache hit rate improvement that directly translated to performance for all other variants with the best-case variant scratch-reuse speeding up by 21% and 16% for linear and autonomous benchmarks respectively over the baseline. Both the performance and hit rate trends are similar for both the benchmarks but since autonomous has more synchronization operations to perform per pipeline stage than the linear, the impact of both performance loss and gain were less pronounced.

8.3.2 Comparison against Spatiotemporal Coherence

Figure 8.9 compares the performance of EntC against STC. We are comparing against the best performing STC with multi-band optimization that grants epochs for up to 4 bands

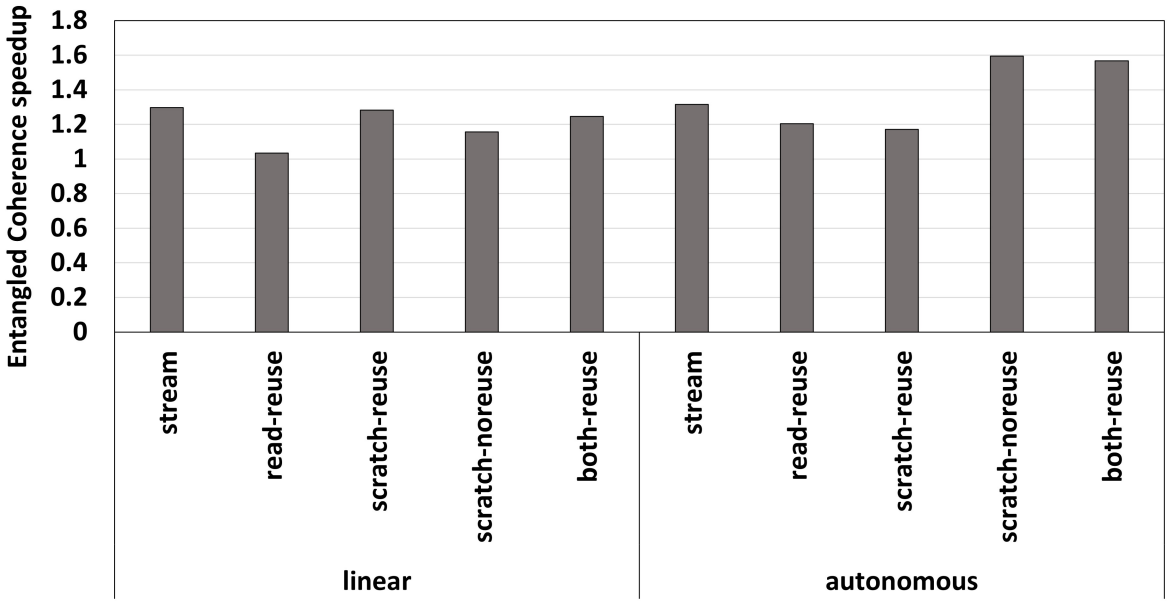


Figure 8.9: Performance comparison against Spatiotemporal Coherence. Across the range, Entangled Coherence performed better than Spatiotemporal Coherence.

simultaneously. Across the range, EntC performed better than STC. There are several reasons that contributed to EntC's performance benefit. The primary reason is that the system we evaluated is a latency-sensitive SoC system (section 6.2.3), and STC is a protocol optimized for latency-tolerant accelerators that can tolerate long store completion latency. For example, the SoC pipeline (emulated by the latency-sensitive GPU) stalls waiting for a write completion because both the linear and pipeline benchmarks have just one wavefront launched per CU. Contrast this to a regular GPU where it can juggle between wavefront contexts to hide long latency memory operations without stalling the pipeline.

The second reason for the performance degradation of STC is due to its inability to grant enough epochs simultaneously to all the CUs. For example, the linear benchmark has 5 pipeline stages with each stage writing to two write bands. That means a total of 10 write bands for this benchmark. Additionally, there is one band for synchronization variables. Thus, at any point, there must be at least 5 write bands and one synchronization

band active. Thus, even assuming the adaptive band optimization was able to correctly find the band boundaries, a multi-band optimization that grants 4 epochs simultaneously cannot satisfy the requirements of 6 bands. It should be noted that these calculations do not include the 5 additional bands for scratch accesses. As a result, the STC's EMU will have to grant epochs every time it wakes up because there will be at least one CU whose epoch is not granted at any given time. Now, since STC can synchronize epochs only across all CUs, granting an epoch for one CU means taking it away from another CU because the total number of multi-bands is fixed to 4. This resulted in 3 orders of magnitude increase in the number of epoch changes for STC compared to EntC. This also contributed to STC's poor performance.

The seemingly simple and straightforward solution of increasing the number of multi-band epochs will not solve this problem either. Suppose we increased the number of multi-bands to maximum number of bands simultaneously accessed by the benchmark. To begin with, this will require benchmark specific changes to an architectural component which is difficult. Even if we get around that issue, since STC synchronizes epochs across all the CUs, all the CUs will have all the bands granted effectively creating a cacheless system because of STC's caching rules. The performance degradation with lower cache hit rate is already discussed previously in Section 8.3.1. While EntC also suffered from poor caching performance for read reuses, it could make up for that loss by providing caching benefits to private/scracth accesses. STC does not have the capability to identify and group private accesses either. Thus, for the system we simulated and for the class of applications we are targeting EntC for, STC cannot match EntC's performance because of EntC's ability to understand the sharing pattern and perform localized epoch synchronization, and also because of its ability to provide caching benefits to private bands.

Although these type of workloads run at fixed frame rates on a system that is provisioned for those frame rates, the variability in the system can lead to some frames falling short of the stringent real-time constraints [59, 68]. A better performing coherence implementation

can help in this situation in two ways: (a) it can reduce the number of frames falling short of the real-time constraints thus reducing the tail latency that is typically used to quantify the performance of these systems [68], and/or (b) it can enable faster frame rate or denser frames with the same frame rate both improving the quality of the output from that system.

8.3.3 Bandwidth and Hardware Cost

After entanglement is established, EntC just has to send one epoch demand request per frame per pipeline stage. The EMU responds to this request with an ack followed by initiating epoch synchronization involving a 4-way handshake. Taking linear benchmark as an example, there will be 10 epoch messages (1 epochDemand request, 1 epochDemandAck, 8 messages for 4-way handshaking with 2 CUs in the entangled group) for 640KB demand accesses per frame (128KB in the read-phase, 128KB reuse-phase, 128KB scratch write and 128KB scratch read in the scracth-phase, and 128KB write phase). Even if we assume all the reuse and scratch phase accesses are filtered by the L1 cache, there will be demand requests and responses to transfer 256KB data between L1 and L2. Additionally, the synchronization accesses will also add to the demand requests and responses. EntC's 10 epoch requests practically add nothing to the bandwidth consumed by the demand accesses for transferring 256KB of data. Thus EntC adds negligible bandwidth overhead.

Table 8.1: Hardware cost of Entangled Coherence implementation

Entanglement map	1 per system hosted by EMU, 32 entry, 44-bits per entry
Band map	1 per CU, 16 entry, 28-bits per entry
BSQ	1 per CU, 64 entry, 32-bits per entry

Table 8.1 lists the hardware cost for implementing EntC. The entanglement map has 32 entries. Each entry stores an epoch address (20-bits band address, 5-bits SEB), 2-bits access type, 1-bit epoch granted status, and two bit-vectors of 8-bits each to store the consumer and producer list. Thus each entry stores 44 bits. The band map is similar but only has 16 entries and 28-bits per entry (producer and consumer lists are not present). The BSQ is

similar to the STC's BSQ but we reduced the number of entries because our baseline GPU only permits one outstanding memory instruction (section 6.2.3). All these structures can be implemented with logic cells and do not require SRAMs.

8.4 Chapter Summary

We implemented, demonstrated the correctness with a working example, and evaluated EntC on a heterogeneous SoC in this chapter (Sections 8.1, 8.2 and 8.3). Our evaluation shows that EntC provides significant performance benefits for latency-sensitive applications that also has scratch/private space accesses (upto 21%) but falls behind the baseline in the degenerate case - that is applications with data reuse but without any scratch/private accesses.

Next chapter provides the implementation and evaluation of Lockstep SC consistency model on an APU SoC.

Chapter 9

Lockstep SC for APUs

There is a plethora of prior work implementing strong models on a CPU [3, 46, 67, 97] or a GPU [50, 87, 96]. However, we implement a strong model for an entire APU SoC with a tightly integrated GPU and CPU. So, we start this chapter with a discussion about how we arrived at this implementation (Section 9.1). Then we proceed with a detailed discussion about our implementation of the Lockstep SC (LSC) model on the baseline APU SoC (Section 9.2) after which we will have a section arguing about the correctness of our implementation (Section 9.3). We conclude this chapter with the evaluation of our implementation (Section 9.4) and a chapter summary (Section 9.5).

9.1 Implementation Considerations

LSC implementation across an entire APU SoC involves implementing strong ordering between dynamically scheduled OoO CPU cores and simple in-order but lockstep executing GPU cores. Since these diverse processing elements represent two ends of the processor architecture spectrum, arriving at a solution that works for both is challenging. Many existing solutions to support strong ordering rely on the speculation capability of CPU cores [44, 45, 86] or on the underlying cache coherency mechanism [3, 87, 98, 100]. However, the GPU cores lack speculation support making them unimplementable on a GPU, and the CPUs and

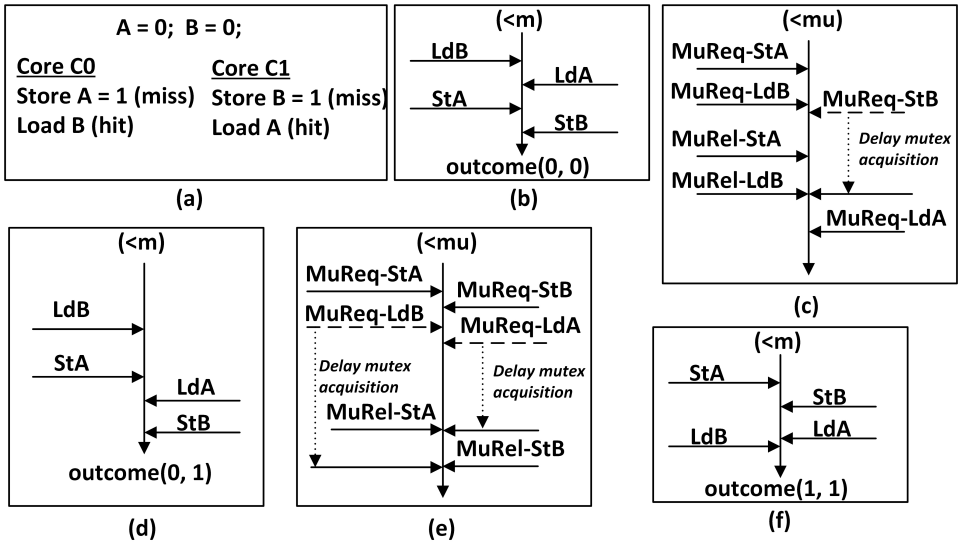


Figure 9.1: Demonstration of atomic SC. (a) Program order of memory operations. (b) Memory order of memory operations. (c) Mutex ordering when LdB beats StB. (d) Memory order with ASC for LdB beats StB. (e) Mutex ordering when StB beats LdB. (f) Memory order with ASC for StB beats LdB. Although atomic SC allows reordering, its mutex acquisition rules ensure the reordering is not visible to other cores.

GPUs have significantly different cache hierarchies making the coherency based methods unattractive as a common solution [11, 37, 49, 80, 98].

Lin *et al.* [67] made the observation that strong ordering can be enforced by preventing reordering of conflicting accesses. In this paper, we leverage this observation for implementing LSC. For preventing the reordering of conflicting access, we leverage the mutex acquisition mechanism proposed by Gope and Lipasti [46]. This mutex mechanism does not require speculation support and does not interfere with the underlying coherence mechanism. Hence, this mechanism can be easily adopted to implement strong ordering on both CPUs and GPUs.

In this section, we first discuss the implementation of strong ordering with mutex mechanism proposed by Gope and Lipasti [46]. Then, we provide a discussion of our proposed mechanisms to implement strong ordering using mutexes on both dynamically scheduled OoO cores and GPU cores.

9.1.1 Strong Ordering with Mutexes

Gope and Lipasti [46] proposed atomic SC (ASC) for implementing sequential consistency on a simple in-order multicore system. ASC allows reordering of memory operations but protects the reordered requests with mutexes making the reordering invisible to other cores. Figure 9.1 provides a simple demonstration of ASC enforcing SC with reordered memory operations. Figure 9.1 (a) shows the program order of memory operations. We assume the stores missed in the cache and loads hit in the cache. Under these assumptions, if cores C0 and C1 allow reordering of younger loads, the loads bypassing the store will create a memory order as shown in Figure 9.1 (b). The outcome (0,0) is forbidden in SC and violates SC.

ASC avoids SC violations by allowing the reordering of operations *iff* the pending operations have acquired a mutex. So, in the previous example, the loads are allowed to bypass stores *iff* both the older store and younger load has acquired a mutex for the addresses they are trying to access. Forcing the reordered instructions to acquire mutexes create a mutex ordering (<Mu) at the mutex pool as shown in Figure 9.1 (c). In Figure 9.1 (c), the mutex request from C0 to address B (MuReq-LdB) beats the mutex request from C1 to the same location (MuReq-StB) forcing MuReq-StB to acquire the mutex only after LdB releases the mutex(MuRel-LdB). Since C0 has acquired mutexes for both the pending store and the younger load under the shadow of that store miss, C0 can now reorder memory operations. Acquired mutexes are released only after the older store (StA) from the same core is retired (StA). Thus, MuRel-LdB is a guarantee that Store A has retired and the load A (load A from C1 will not be a cache hit because StA will invalidate all stale copies) from C1 will see that store. The memory order forced by ASC is shown in the Figure 9.1 (d) and the outcome (0, 1) is SC.

Figure 9.1 (e) shows a scenario in which MuReq-StB beats MuReq-LdB to the mutex pool and the MuReq-LdB is delayed at the mutex pool. In this situation, since LdB has not acquired its mutex, it cannot bypass StA creating a memory order given in Figure 9.1 (f). The outcome(1,1) from this execution is still SC. Since ASC allows reordering iff all the

participating operations have acquired a mutex, the reordering is not visible to other cores because all the reordered instructions become visible as if they have executed *atomically*. A formal proof showing that ASC enforces SC can be found elsewhere [46].

9.1.2 Extending Mutex Mechanism to OoO Cores

Dynamically scheduled OoO cores buffer stores that are completed (committed) but waiting for retirement in store buffers. From the memory system perspective, a store is not completed until it is retired. So, for implementing strong ordering with the mutex mechanism, all stores that are waiting in the store buffer and all younger memory operations that are under the shadow of these stores should acquire mutexes before allowing them to be reordered. However, OoO cores issue memory operations out of order to the memory system. But since those operations are considered speculative till they are committed in program order by ROB, enforcing the mutex acquisition requirement at the time of issuing a speculative memory operation is not necessary. The mutex acquisition requirement need to be enforced only for allowing a memory operation to be moved from speculative to non-speculative state, that is at the instruction commit.

As such, when a memory operation reaches the ROB head and there are pending stores in the store buffer, that instruction is allowed to commit only when that instruction and all the pending stores in the store buffer have acquired a mutex. Since mutexes are released only after the store miss is serviced, just verifying that the youngest store in the store buffer has its mutex guarantees that all pending stores in the store buffer have acquired mutexes.

9.1.3 Extending Mutex Mechanism to GPU

LSC does not enforce ordering among the requests generated by the same wavefront instruction. With this ordering relaxation, extending mutex mechanism to GPU is straight forward. When there are pending stores from a wavefront, those pending store requests and all younger memory requests from that wavefront should acquire mutexes before issuing

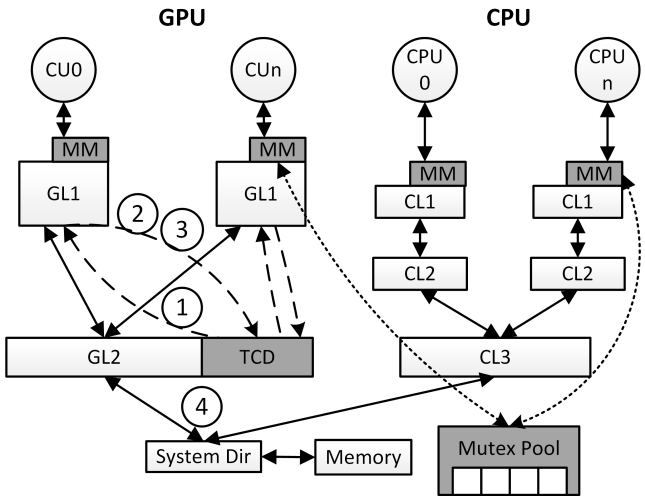


Figure 9.2: APU system with changes to support Lockstep SC. The blocks added to the baseline are shown in gray. The dotted lines are the extra messages needed for LSC. The extra messages include the invalidation message from tagless coherence directory (TCD) to the sharer GL1s ①, invalidation ack and eviction notification from the GL1s to TCD②③, the forwarded eviction notification to the system directory④, and the mutex requests and responses between the GPU and CPU mutex managers (MM) and the mutex pool.

these younger requests to the memory subsystem. However, since all modern GPUs coalesce requests to the same cacheline, the mutex requirement is often drastically reduced. In the best case, when all requests can be coalesced to the same cacheline, just one mutex can satisfy this requirement for all memory requests from the same wavefront instruction.

9.2 Implementation

This section discusses the implementation of LSC. Figure 9.2 shows the modifications needed to the baseline APU system to implement LSC. We start with changes to the GPU memory system.

9.2.1 Changes to GPU Memory System

The baseline GL1 is made coherent with HRF synchronization operations [54]. Since LSC does not have these synchronization operations, GL2 forwards the invalidation messages from the directory to GL1 to make it coherent ① as shown in Figure 9.2. To avoid broadcasting these invalidation messages to all GL1s in the GPU, the GL2 keeps track of the sharers with a tagless coherence directory (TCD) [108]. TCD is not required for LSC (any conventional directory will work), but it is an appealing solution for scalability and other reasons [108].

Non-silent Evictions: Since TCD is tracking GL1 sharers, eviction notification should be sent to TCD from GL1s. However, since TCD keep sharer information as hashes stored in sharing vectors, the evicted address alone is not sufficient to reset the appropriate bits in the sharing vector. After each eviction, the hash functions for all remaining blocks in the set are calculated in a typical TCD [108]. This recalculation is relatively simple if the GL2 is inclusive, because an inclusive GL2 can track the entries of a sharer from which the hash can be recalculated [108]. However, since GL2 in our implementation is non-inclusive, the GL1 on evicting a block calculates the hash functions for all remaining blocks in that set and sends the recalculated sharing vectors with the eviction messages. The recalculation of the hash requires reading the entire tag array of a set after an eviction and generating a 320 bit (five 64-bit sharing vectors) eviction message. Since we read the entire tag array of a cache set as part of normal cache operation before evicting a cache line, the tag array read to recalculate hash can be amortized. However, sending 320 bits per eviction message is expensive. So, to reduce the number of eviction messages transmitted between GL1s and GL2, we only send the recalculated hash once every N evictions from that set. A higher value of N increases the false positives whereas a lower value of N will result in high eviction message bandwidth. To count the number of evictions since the last hash recalculation, a $\log_2$N-bit counter is added per set to the GL1. This counter is incremented each time a cacheline is evicted from that set and the N^{th} eviction will generate an eviction message with the recalculated hash. The GL2 on receiving this recalculated hash will update the corresponding TCD entry.

Figure 9.2 illustrates the operation of this GPU memory system implementation. On receiving a write-through request from GL1 or an invalidation request from SD, GL2 consults the TCD to see if there are any sharers for that cacheline and forwards invalidation message to the sharer GL1s ①. The GL1s invalidate the data and respond to the invalidation message with an invalidation ack ②. In the event of a false positive by TCD, the GL1 will receive the invalidation request in I state but it still responds with an ack. By making the GL1s coherent with invalidation messages, the entire APU system now preserves the single-writer invariant. Finally, when GL1 evicts a block, the eviction message is forwarded to the GL2 ③. If the eviction resulted in removing that block from all GL1s, the eviction message is forwarded to the SD ④ and the probe filter in SD is made inclusive of all caches in the APU.

9.2.2 Changes to CPU Pipeline

LSC requires minimal changes to the CPU pipeline. On a store miss, the CPU mutex manager (CMM) notifies the ROB about this store miss. A memory instruction under the shadow of a store miss is allowed to commit only after (a) the missed store has acquired a mutex and (b) the committing memory instruction has acquired a mutex. So, ROB commits an instruction only after all pending stores in the store buffer and the memory instruction at the head of ROB have acquired a mutex. Since mutexes are released only after a pending store is retired, ROB has to only check if the youngest store in the store buffer and the committing instruction have acquired mutexes.

If the store buffer has pending stores, the ROB issues a mutex request for the memory instruction at the ROB head. The CMM forwards the mutex acquisition ack to ROB. On receiving a mutex ack, the ROB commits that memory instruction if the youngest pending store and the head instruction has acquired mutexes. With this implementation, a younger store under the shadow of a store miss is moved into the store buffer only after acquiring a mutex. The changes to the ROB are trivial. A 2-bit flag to indicate waiting on mutex ack (one bit for the head instruction and one bit for the store) and a check to see if there are

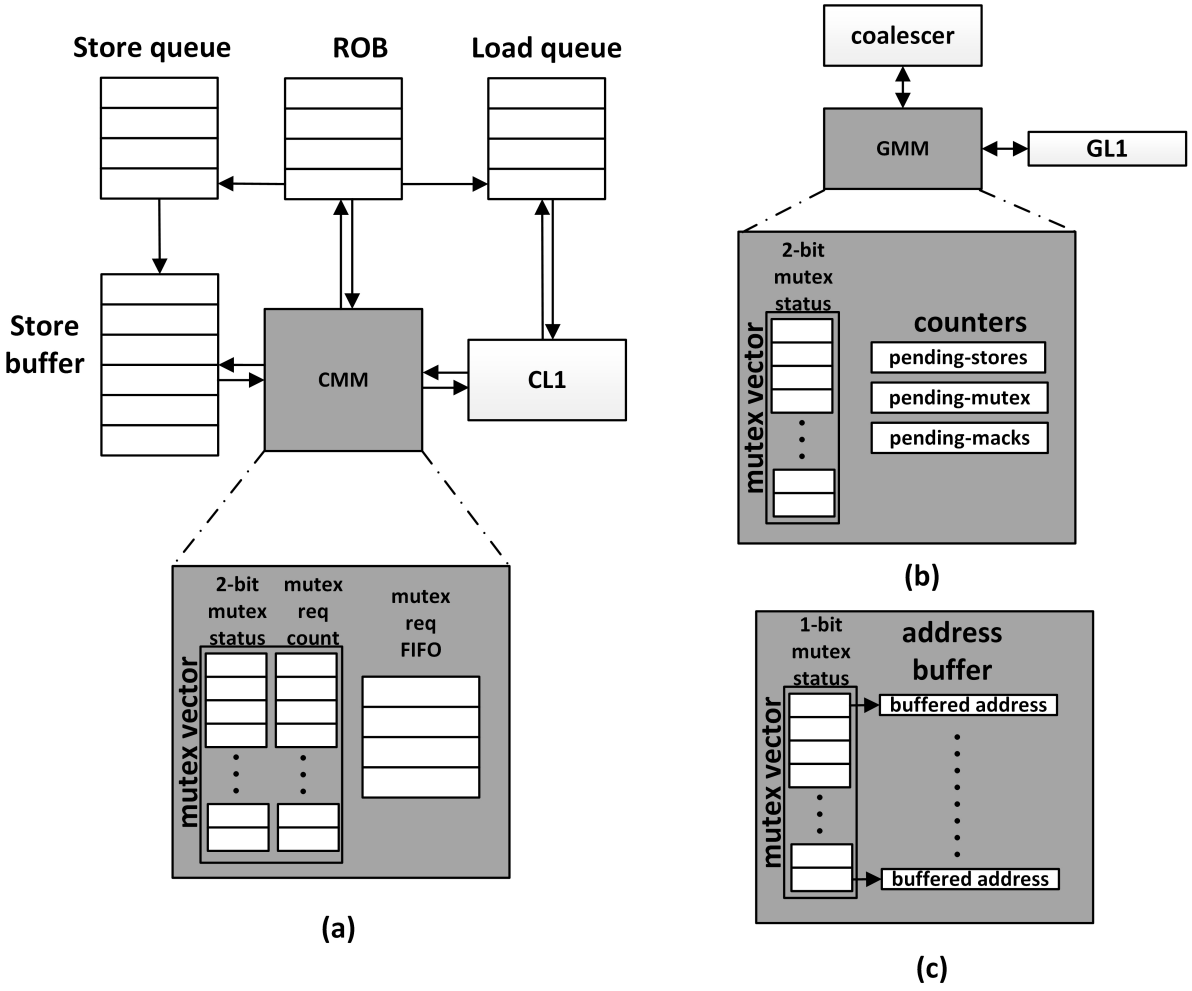


Figure 9.3: Mutex manager and mutex pool. (a) CPU mutex maanger interacting with other CPU core components. (b) GPU mutex manager interacting with CPU components. (c) Mutex pool with mutex vector and address buffers. The mutex vector of both CPU and GPU mutex managers tracks the acquired and requested mutexes. The mutex vector in the mutex pool keeps track of the acquired mutexes. It also buffers pending mutex requests in the address buffer.

pending stores in the store buffer. Next, we discuss the implementation of CMM.

9.2.3 CPU Mutex Manager

The CPU Mutex Manager (CMM) is responsible for requesting mutexes, storing the acquired mutexes and releasing mutexes. Figure 9.3 (a) shows the components inside a CMM. CMM has a mutex vector to keep track of the requested and acquired mutexes. This mutex vector is indexed by a hash function generated from the cacheline address of memory requests. In our implementation, we use $\log_2$ (mutex pool size) lower bits of the cacheline address as the hash function. Each entry in the mutex vector has 2-bits indicating the status of that mutex. The status of a mutex can be *mutex-requested*, *mutex-acquired* or *mutex-invalid*. A *mutex-requested* state indicates that a request to that mutex is pending. A *mutex-acquired* indicates that this core has exclusive lock on that mutex and *mutex-invalid* is the default state. Each mutex vector entry also has a request count field that keeps track of the number of pending mutex requests to that mutex.

The operation of CMM is shown in Figure 9.4. The store buffer issues a store to cache ① and the cache notifies if the store is a miss. If the store is a miss and there are no pending store misses, then that store is added to the Mutex Request FIFO (MRF) ② and the MRF issues a mutex request to mutex vector ③. The Mutex vector indexes into the mutex entry corresponding to the store address and issues a mutex request to the mutex pool (mutex pool is described in detail in section 9.2.5) if the status of that mutex is *mutex-invalid* ④⑤. After issuing the request, this status is changed to *mutex-requested* and the mutex request count is incremented. Since there are stores pending in the store buffer, the ROB now issues a mutex request to MRF for all younger memory requests reaching ROB head ⑥ and does not allow that instruction to commit until it receives an acknowledgement for that mutex request. The MRF forwards this request to mutex vector③. The requested mutex may be already available in the mutex vector. If not, a new mutex request is issued following steps ④⑤. If the requested mutex is available in the mutex vector or when the CMM receives the mutex

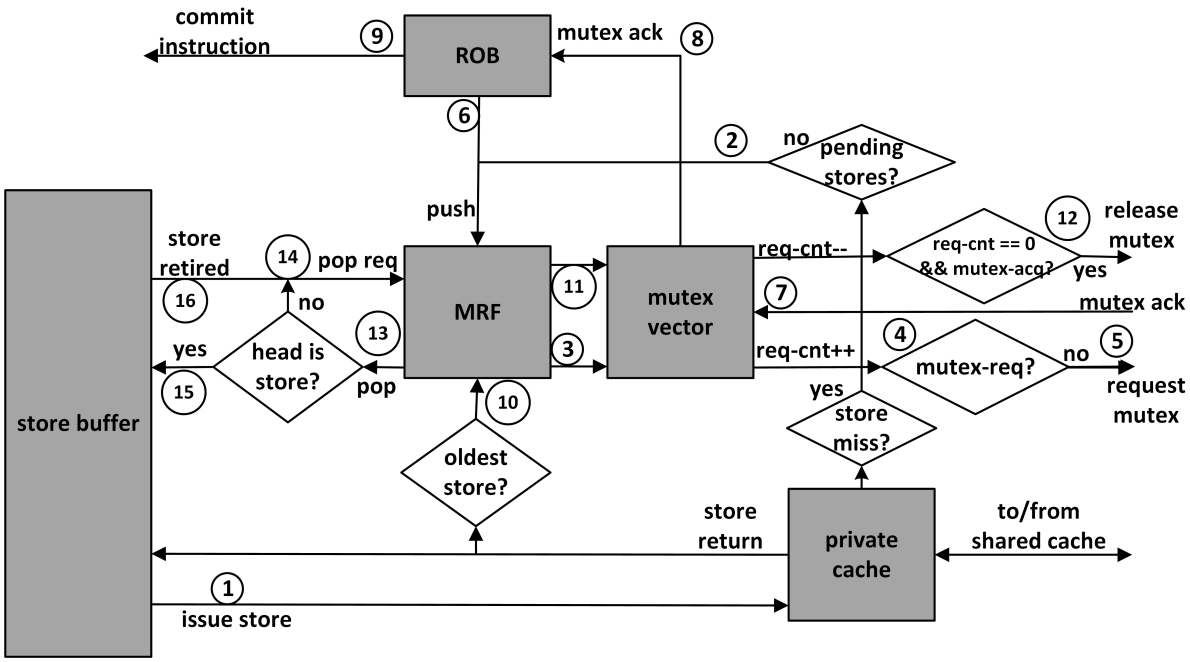


Figure 9.4: CPU mutex manager operation. The CPU mutex manager requests, caches and releases mutexes.

acknowledgement for these mutex request at a later time ⑦, the mutex acknowledgement is forwarded to ROB ⑧. The mutex status is also set to *mutex-acquired* when a new mutex acknowledgement is received. If ROB receives mutexes for both the youngest pending store and the instruction at ROB head, it commits the instruction⑨.

When the oldest store returns ⑩, the MRF pops its head entry and forwards the mutex release request to mutex vector ⑪. Mutex vector decrements the request count and checks if the status is *mutex-acquired* and request count is 0. If this check passes, the mutex is released ⑫ after changing the mutex status to *mutex-invalid*. After popping the store ⑬, the MRF checks if the next instruction at FIFO head is a store. If the head is not a store, the process of popping requests and releasing mutexes continues ⑭. If the head is a store ⑮, then the store buffer is consulted to see if that store is retired. If that store is retired ⑯, then the process of popping requests and releasing mutexes continue till the MRF is completely drained or the head is a store that has not retired yet. When MRF is completely drained,

there will be no pending store in the store buffer and ROB resumes its normal operation.

Apart from simple condition checking hardware, CMM just adds a FIFO (MRF) and a direct mapped mutex vector. Thus, this implementation of CMM avoids any expensive associative structures.

9.2.4 GPU Mutex Manager

Similar to CMM, GPU Mutex Manager (GMM) manages the mutexes for CUs. However, GMM manages mutexes per wavefront. Figure 9.3 (b) shows the per wavefront structures in a GMM. The GMM has a *pending-store* counter that keeps track of the outstanding store requests from that wavefront. A *pending-mutex* counter keeps track of the outstanding mutex requests from that wavefront. The mutex vector in GMM only has a 2-bit status field. When stores from a wavefront instruction reach the GMM, these stores are issued to the L1 cache. Since the GPU cache is write-through, all stores are considered to be store misses and a mutex request is issued for every store. Both *pending-mutex* and *pending-store* counters are incremented to keep track of the outstanding mutex and stores respectively. However, before issuing a mutex request, the mutex vector is consulted and a request is issued only when the status is *mutex-invalid*.

Unlike CPU mutex requests, the mutex pool (mutex pool is described in detail in section 9.2.5) does not buffer GPU mutex requests because of the limited buffering space available with it. So, the mutex pool can send back a GPU mutex response with a negative acknowledgement (nack) indicating that the request did not acquire a mutex. When *pending-mutex* reaches zero and all these mutex requests have acquired their mutex, the next wavefront instruction is serviced by GMM. The acquired mutexes are released when the *pending-store* counter reaches zero.

Section 5.3.3 discussed implications of LSC on coalescer design. The coalescer issues coalesced requests from the same wavefront in program order to the L1 cache. The mutex acquisition mechanism discussed above allows younger requests to be issued under pending

stores without making reordering of conflicting accesses visible to a wavefront. However, for the implicit flag synchronization discussed in section 5.2.2, the younger loads should see the value of older stores from the same wavefront. Unlike CPUs, GPUs do not employ a store buffer. Also, the L1 cache in our implementation blocks a younger load under a pending store to the same address. Thus, a younger load sees the value of older store from the same wavefront and LSC's implicit synchronization flag for lockstep threads is guaranteed by our implementation.

9.2.5 Mutex Pool

The mutex pool keeps track of the mutexes acquired by a CU or a CPU core with a direct-mapped mutex vector. The direct mapped mutex vector enables the mutex pool to be implemented as an address interleaved, distributed structure making it extremely scalable. It is indexed similar to the mutex vectors in CMM and GMM. It has a single bit to store the mutex status and an address buffer to buffer pending requests to that mutex as shown in Figure 9.3 (c). If the bit is set, it indicates that mutex is already acquired by a core in the APU. If a mutex request arrives at mutex pool, the mutex pool checks if that mutex is available by checking the status bit. If the status bit is not set, that mutex is available and the mutex request is acknowledged by mutex pool after setting the status bit. If the status bit is set, mutex pool buffers the request into a per mutex address buffer only if that mutex request is from a CPU. If the mutex request is from a GPU core, the mutex pool responds with a negative acknowledgement (nack) indicating the requester did not acquire that mutex. Since a CPU core does not issue multiple outstanding requests to the same mutex, the address buffering needed per mutex vector entry is bounded by the number of CPU cores in the system. However, since GPU cores keep track of requests at wavefront granularity and there are 320 wavefronts (40 wavefronts per CU and eight CUs) in our implementation, the buffering space needed to buffer mutex requests from the GPU makes GPU mutex buffering unattractive. Additionally, CPUs are latency sensitive and this design avoids CPU requests

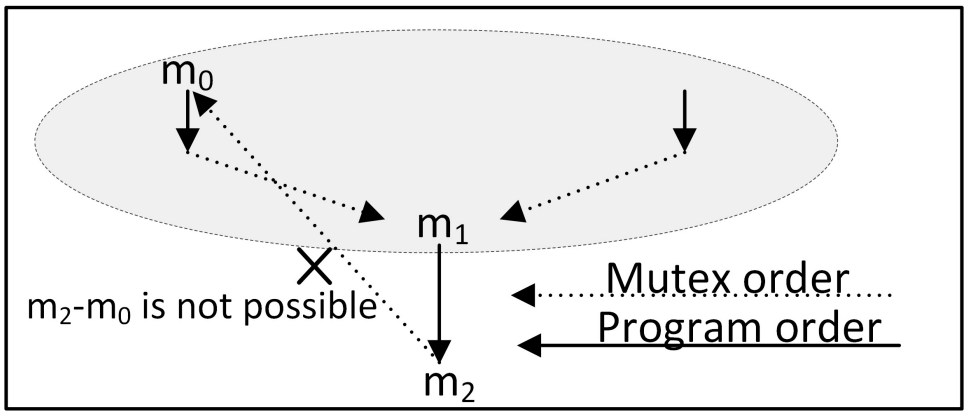


Figure 9.5: Constraint graph demonstrating the correctness of Lockstep SC implementation. Mutex acquisition rules prevent cyclic m_2-m_0 edge.

waiting behind GPU requests. Since GPUs are inherently latency tolerant, the nacking of its requests will have little impact on its performance. When a mutex release request reaches the mutex pool, the pending address buffer is consulted and mutex is transferred to the next requester waiting in the pending address buffer. If the pending buffer is empty, the mutex status bit is unset and that mutex becomes available for future requests.

Scalability of the Mutex Pool

To increase the scalability, the mutex pool can be implemented as an address-interleaved distributed structure. Each address will still have a unique mutex vector entry in this distributed structure and thus the conflicting accesses will map to the same unique entry providing exclusion. Thus distributed mutex pool functions identically to a centralized mutex pool but with better scalability.

9.3 Correctness of the Implementation

Since LSC enforces ordering constraints on only MOp1 $<_p$ MOp2, proving that our implementation enforces MOp1 $<_m$ MOp2 ordering for MOp1 $<_p$ MOp2 is sufficient. Gope and

Lipasti [46] have proved this for the CPU implementation using the constraint graph method from Lin *et al.* [67]. Since the mutex acquisition rules are the same for our CPU implementation and the ASC implementation by Gope and Lipasti, we only extend that proof to the GPU implementation. Figure 9.5 shows the constraint graph used by Lin *et al.* pred(m_1) is a set of memory operations (including m_1) that appear before m_1 in PO. Committing any memory operation m_2 will guarantee SC, provided all pred(m_1) that conflict with m_2 have completed. That means if committing m_2 does not create a conflict order edge and result in a cyclic constraint graph, then committing m_2 is safe. Conflict edges can be represented by mutex edges as shown in Figure 9.5. For GPUs, m_2 represents the *set of memory requests* generated by a memory operation. Memory operation m_1 is the store instruction that generated pending stores. Since our implementation allows m_2 to proceed only after acquiring mutexes for all pending stores from m_1 and all requests from m_2, we argue that all memory operations that conflict with m_2 have retired (and released mutexes) without which (a) m_2 will not be able to acquire mutexes in the first place. Thus, the mutex edge from m_2 to m_0 can never be created with the GPU mutex acquisition rules and hence our GPU implementation enforces SC for operations that obey MOp1 $<_p$ MOp2.

9.3.1 Deadlocks and Mutex Acquisition

The mutexes are acquired in program order by both the CPU and GPU. For the GPU, the younger memory operations are allowed to request a mutex only after all memory operations from an older wavefront instruction has acquired a mutex. Thus, the mutex acquisition is in program order in a GPU. In a CPU, although the instructions are executed out-of-order, the mutex acquisition itself is delayed to the commit stage and hence the mutex acquisition is in program order. This in-order acquisition of mutexes prevent any younger instruction from holding on to a mutex while an older instruction is waiting for the same mutex. Thus, cyclic mutex dependencies never occur with this design avoiding any possibility of deadlock. Hence, our implementation is deadlock free.

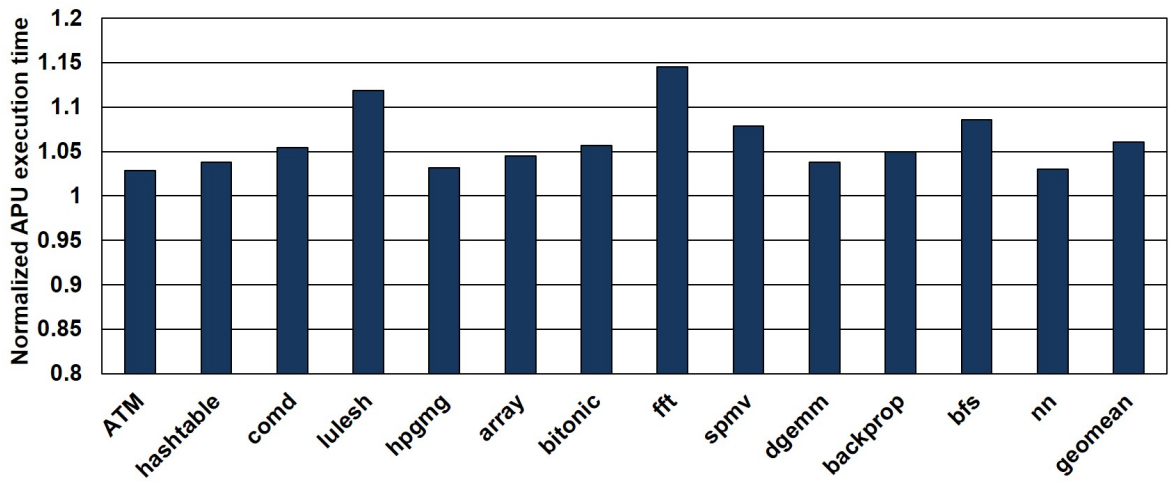


Figure 9.6: Normalized application execution time. Lockstep SC implementation was slower than the baseline weaker consistency model by 6.11% across all evaluated workloads.

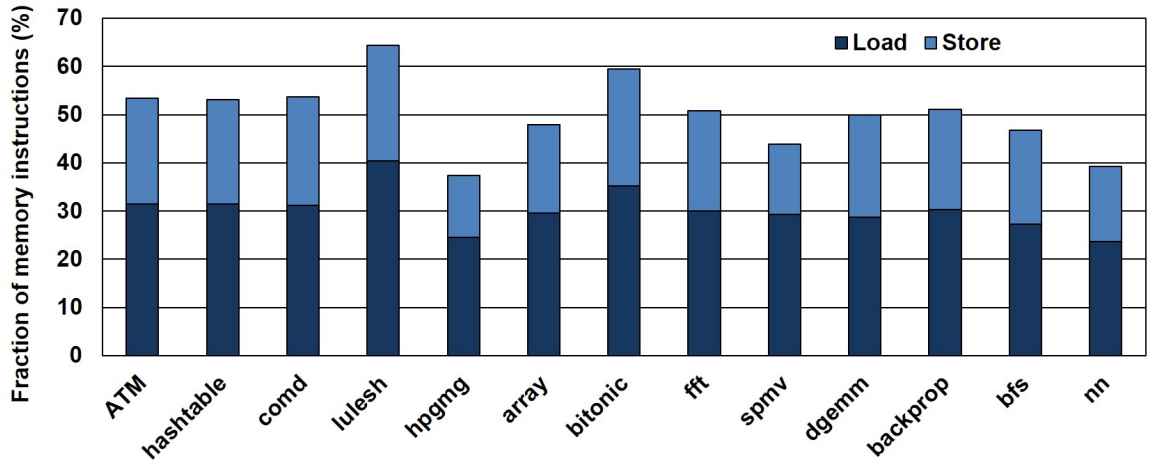


Figure 9.7: Fraction of memory instructions committed by each benchmark. The figure also shows the load-store split.

Table 9.1: Lockstep SC specific simulation parameters

MRF	100 entries
Mutex Pool	1024 entries, 30 cycles away
Tagless Directory	5 hash functions, 64-bit sharing vector

9.4 Evaluation

We evaluate LSC in this section. First we evaluate the performance of LSC compared to the baseline. We evaluate the GPU variant with 16KB L1 cache. LSC specific simulation parameters can be found in the Table 9.1.

9.4.1 Application Performance Evaluation

We compare the performance of LSC system against the baseline. Since the baseline adheres to a weaker model than LSC, our aim is to show that a strong consistency model like LSC can achieve performance comparable to this baseline.

Figure 9.6 shows the execution time of LSC compared to baseline for various benchmarks. The LSC was slower than a weaker consistency model by 6.11%. However, from that figure, it can be seen that LSC was performing within the 5% range for more than half of the evaluated benchmarks. The figure shows that two benchmarks *lulesh* and *fft* slowed down by more than 10% with weaker models.

To understand this performance difference, we first looked into the CPU execution of these applications and analyzed their instruction composition. Figure 9.7 shows the fraction of memory instructions committed by each benchmark. It can be seen that most of these benchmarks have more than 50% memory reference instructions. That figure also shows the percentage of loads and stores. Although LSC adds mutex acquisition latency to the commit stage of a memory instruction, an OoO execution pipeline is usually not affected by it because of its ability to extract ILP over a wide instruction window. As such, LSC will stall the execution pipeline only if the ROB becomes full. However, for memory intensive applications, LSC will frequently stall commit preventing the memory instruction at the head

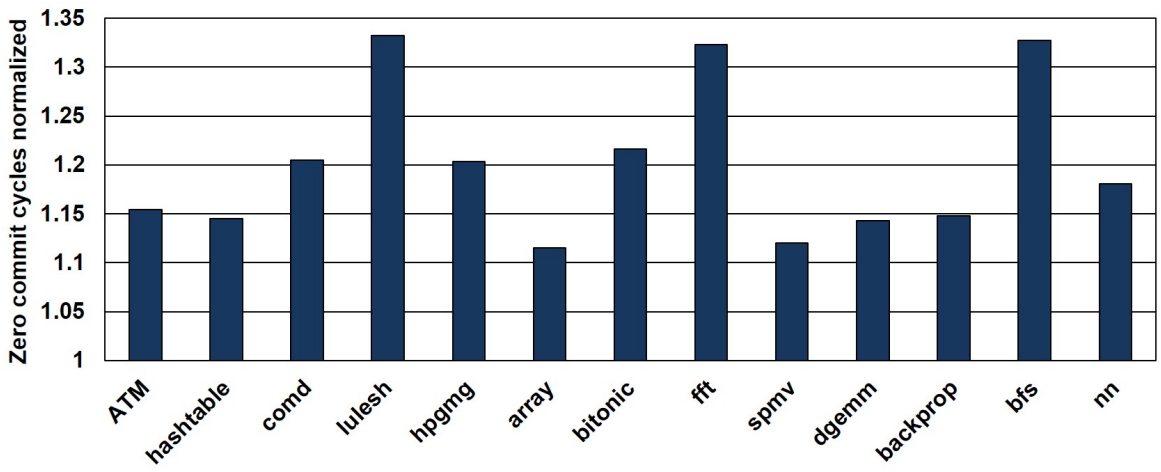


Figure 9.8: Increase in zero commit cycles. This measures the commit stage stalls introduced by Lockstep SC.

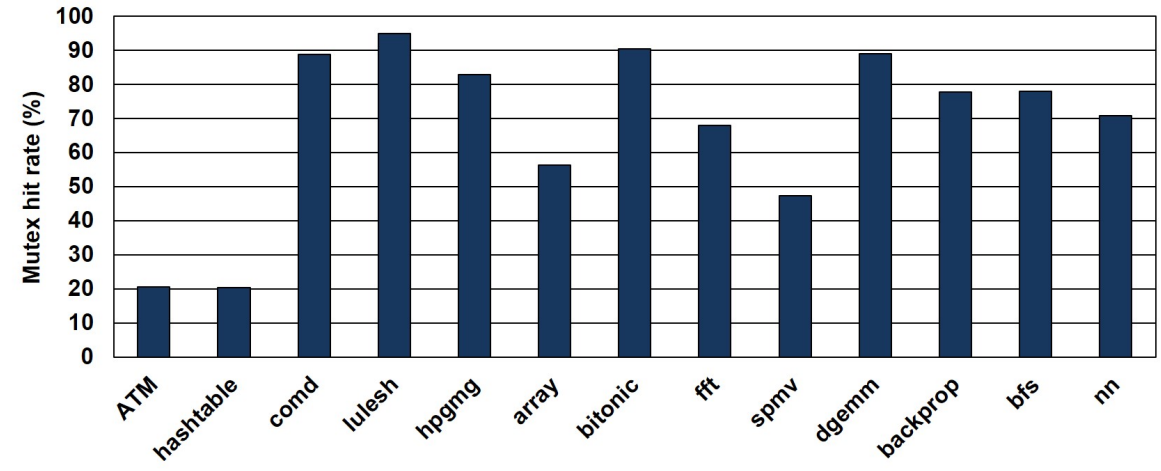


Figure 9.9: Mutex hit rate. It quantifies the reuse of mutexes by a CPU core. Higher hit rate reduces the mutex acquisition latency and increases performance.

of ROB from committing and freeing up ROB entries and potentially hurting application performance after the ROB gets full.

Figure 9.8 shows the increase in number of zero commit cycles with LSC. Zero commit cycle indicates the number of cycles in which no instruction was committed. It is a measure of the commit stage stall cycles. However, it should be noted that the commit stage stalls did not directly translate to performance loss because the OoO execution pipeline will be stalled only if ROB gets full.

Even if an application is memory intensive, the mutex acquisition latency will affect the critical path in the commit stage only if the head instruction has not acquired a mutex. Typically, the memory instructions of an application have high spatial and temporal locality, so the mutexes acquired by the core can be reused by multiple memory instructions. Figure 9.9 shows the mutex hit rate of different benchmarks. While most of the applications have high hit rate, *fft* had relatively low hit rate and as such did not benefit from mutex reuse. It can be also seen from that figure that *lulesh* observed a very high mutex hit rate and that helped *lulesh* to perform better than *fft* although it has a larger fraction of memory instructions than *fft*. *hashtable* and *ATM* only have their GPU phases simulated, and the CPU phase hardly had any work to do other than launching kernels; hence the low mutex reuse.

Our implementation is quite conservative: for example, we do not attempt to prefetch mutexes in the OoO core (analogous to exclusive prefetching of write permission [42]). Another possible optimization is caching of mutexes for reducing the mutex acquisition latency overhead. Adding these optimizations would complicate implementation, but is likely to recover these performance losses. We leave exploration of these opportunities to future work.

GPU Performance Comparison

Figure 9.10 compares GPU execution time of LSC against baseline HRF model. Across the spectrum of benchmarks, LSC performs very close to the HRF baseline with benchmarks slowing down by only 0.76% with LSC. The latency tolerant GPUs can hide the mutex acqui-

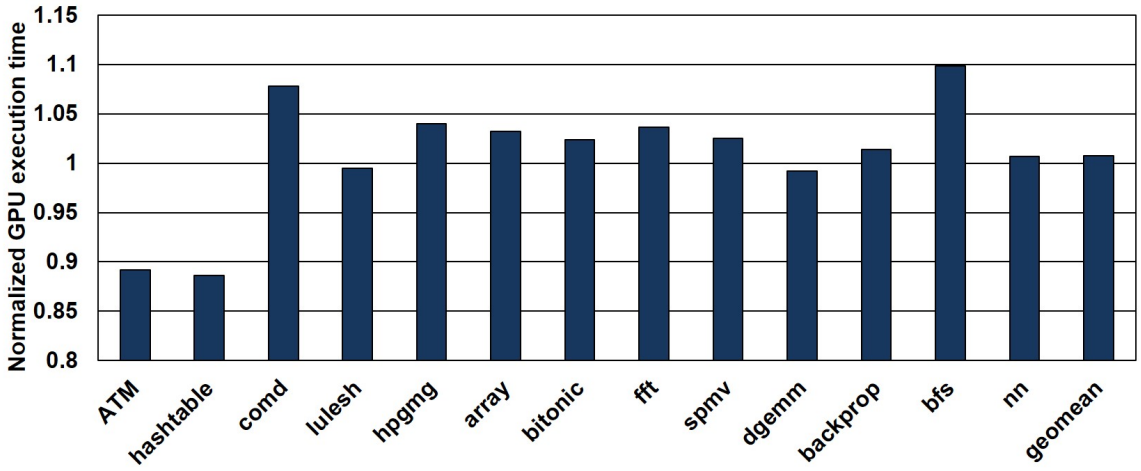


Figure 9.10: Normalized GPU execution time. Lockstep SC implementation was slower than the baseline HRF model by 0.76% across all evaluated workloads. However, fine-grain data sharing benchmarks *hashtable* and *ATM* performed better with Lockstep SC because compared to the baseline, it avoided the GPU L1 invalidation at synchronization boundaries.

sition latency by overlapping execution with context switching between multiple wavefront contexts. Because of this, mutex acquisition latency will not show up on the critical path of GPU execution, and a strong model like LSC performs well on a GPU.

Figure 9.10 also shows that *hashtable* and *ATM* performed better with LSC. Since these benchmarks have fine-grained data sharing, an HRF model does an acquire operation (that is invalidating GPU L1 cache lines) before attempting each insertion (for *hashtable*) or transaction (for *ATM*) to get the latest updates. The LSC model maintains data updates with invalidation messages provided by the underlying hardware coherence mechanism and as such does not encounter the performance penalty of invalidating the entire GPU L1 cache. Consequently, the GPU L1 hit rate of *hashtable* increased from 10.12% to 40.50% and that of *ATM* increased from 40.40% to 66.37% with LSC compared to the baseline. This resulted in *hashtable* and *ATM* achieving a speedup of 12.81% and 12.14% respectively.

Finally, Figure 9.11 shows the performance penalty of implementing HRF with hardware coherence mechanism. To measure this overhead, we created an intermediate baseline by

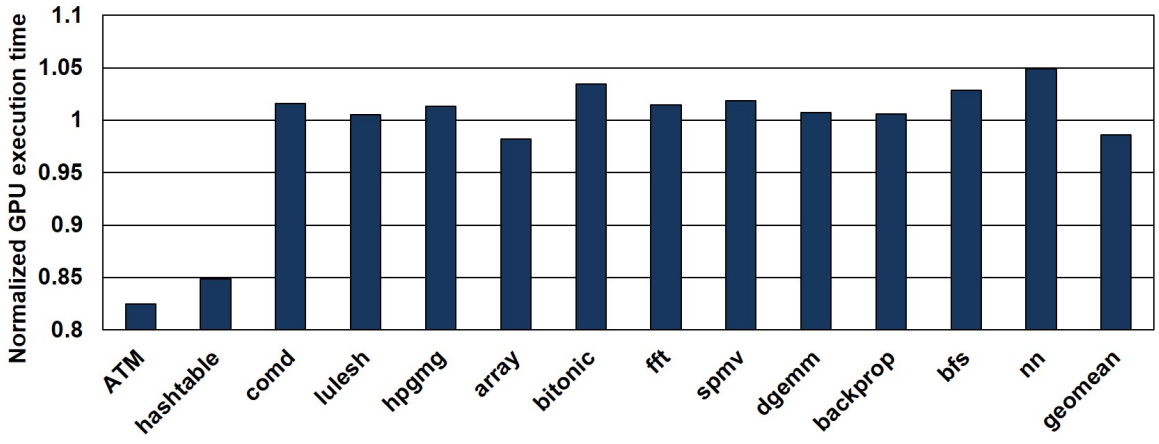


Figure 9.11: Hardware coherence overhead. Performance penalty of implementing HRF with hardware coherence. All benchmarks observed slowdown except *hashtable* and *ATM*.

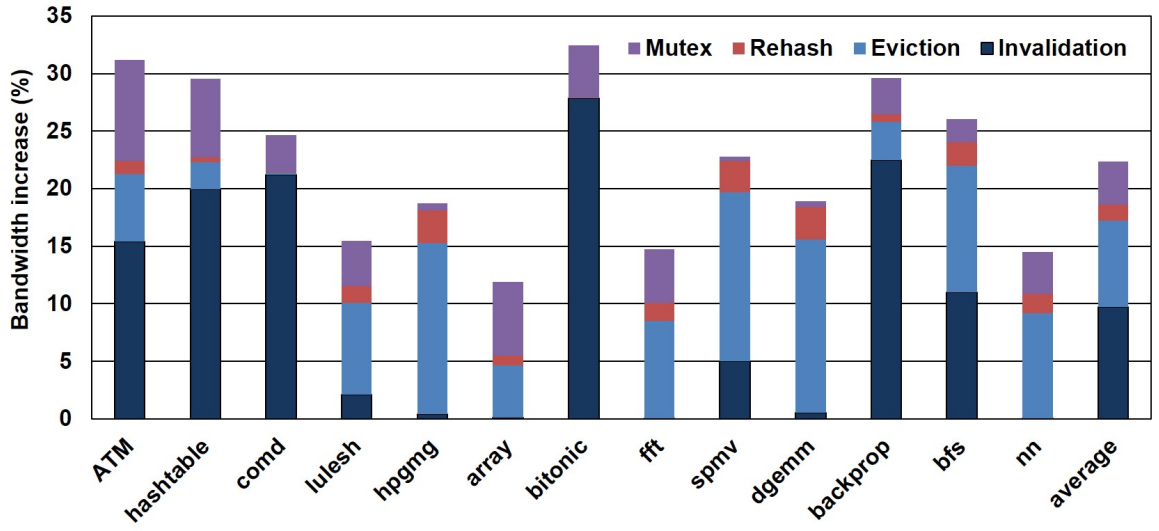


Figure 9.12: Bandwidth increase with Lockstep SC. Bandwidth consumption increased by an average of 20.94% with eviction, invalidation and mutex traffic contributing 9.70%, 7.51%, and 3.73% respectively.

modifying the baseline APU system that broadcasts invalidation messages to the GL1 to make it read coherent. Thus, an acquire operation does not need to invalidate the entire GL1 in this implementation. From that figure, it can be seen that making GL1 coherent incurred a performance penalty for all evaluated benchmarks except *hashtable* and *ATM*. For these two benchmarks, the performance improved by 17.82% and 21.20% respectively.

9.4.2 Increase in Bandwidth Consumption

LSC modifies the GPU memory subsystem to send invalidation and eviction messages between GL1 and GL2. The bandwidth increase by this extra traffic is given in Figure 9.12. The mutex acquisition and release traffic is given as *Mutex* in that figure. The traffic generated by eviction messages and its acknowledgement is given as *Eviction*. *Invalidation* measures the bandwidth generated by the invalidation messages and its acknowledgement. The recalculated hash generated every 16^{th} eviction is shown as *Rehash*. For benchmarks like bitonic sort that shares data across CUs, the extra bandwidth is dominated by *Invalidation* traffic. The streaming nature of GPU applications lead to *Eviction* messages and most of the benchmarks contributed to eviction traffic as shown in Figure 9.12. The figure also shows that *Rehash* traffic has contributed little to the bandwidth consumption. Overall, the GPU bandwidth requirement increased by an average of 22.35% with eviction, invalidation and mutex traffic contributing 9.70%, 7.51%, and 3.73% respectively.

LSC implementation did not modify baseline CPU memory subsystem and the only addition to the CPU traffic was the mutex acquisition and release traffic. Since CPU has a very high mutex hit rate (Figure 9.9), only few memory operations had to request mutexes and the mutex traffic only contributed 0.42% to the CPU bandwidth consumption.

9.4.3 Sensitivity to Mutex Request FIFO size

We ran experiments with 400 entry MRF and did not see any noticeable performance improvement indicating that a 100 entry MRF (default MRF size in our experiments)) captures

all the reordering opportunity for memory accesses.

Table 9.2: Hardware cost of Lockstep SC implementation

Mutex vector	1024 entries; 5 bits (CPU core) or 2 bits (CU) per entry
MRF	100 entries per CPU core
Mutex pool	1024 entries; 1-bit and one address buffer per entry
TCD	5 64-bit hash functions per set per core
GPU L1	4 bits per set to count 16 evictions

9.4.4 Hardware Cost

Table 9.2 lists the hardware cost of implementing LSC. All structures added by our implementation are direct-mapped and do not require expensive associative lookups, and these structures are extremely small compared to other CPU or CU resources. For example, the direct-mapped mutex vector at CMM is only 640 bytes (5 * 1024 bits). Compared to our baseline 64KB set-associative CPU L1, CMM mutex vector is only a fraction of its size and adds negligible area overhead to the baseline.

9.5 Chapter Summary

In this chapter, we described an efficient LSC implementation on an APU SoC (Section 9.2). We also provided a formal proof of our implementation in Section 9.3. Evaluation of our implementation in Section 9.4 showed that the geometric mean performance cost for Lockstep SC is just 0.76% for GPU execution and 6.11% for the entire APU SoC compared to a baseline with a weaker memory consistency model. We were able to achieve this performance with an implementation that adds only extremely small direct-mapped structures compared to other CPU or GPU structures. Thus, we demonstrated that a strong model like Lockstep SC can be efficiently implemented on an APU SoC without expensive hardware structures.

The next chapter concludes this dissertation.

Chapter 10

Conclusions

In this chapter, we first provide a concluding summary of the work presented in this dissertation (Section 10.1). Then we list few directions for expanding this research in the future directions section (Section 10.2). Finally, I will reflect on this work and list my thoughts on this expanding landscape of heterogeneous SoCs with diverse accelerators (Section 10.3).

10.1 Summary and Conclusions

We introduced spatiotemporal coherence, a novel low overhead cache coherence protocol that also provides write atomicity by establishing mutual agreement among the processors about the modifiable addresses at any given time. We motivated the need for a strong model on GPUs and a low overhead coherence mechanism that is capable of supporting these strong models by demonstrating the programmability, portability and performance limitations of the existing heterogeneous-race-free model. Realizing that a formal strong model does not exist for lockstep executing threads, we defined and formalized the lockstep sequential consistency model with ordering rules for lockstep execution. We outlined approaches to improve the performance of spatiotemporal coherence including proposing a variant called Entangled Coherence that is suited for latency-sensitive applications with fixed sharing pattern.

Finally, we implemented and evaluated (a) spatiotemporal coherence along with its opti-

mizations on a GPU system, (b) Entangled Coherence on an emulated heterogeneous SoC, and (c) lockstep sequential consistency on an APU SoC. Our evaluations show:

- Spatiotemporal coherence can outperform the baseline for benchmarks with data reuse across kernel boundaries while providing comparable performance as the baseline for other benchmarks.
- Entangled Coherence provides significant performance benefits for latency-sensitive applications that also has scratch/private space accesses but may fall behind the baseline in the degenerate case - that is applications with data reuse but without any scratch/private accesses.
- A strong model like lockstep sequential consistency can be efficiently implemented in an APU SoC without expensive hardware structures.

We conclude that Spatiotemporal Coherence is a well-performing GPU cache coherence protocol. It has all the desired characteristics of software-managed coherence while providing all the benefits of implicit hardware coherence. As such, it is an attractive protocol to support strong models on GPUs. Its extension, Entangled Coherence, is a suitable candidate for latency-sensitive scenarios with fixed sharing pattern across processing cores such as autonomous driving pipelines and mobile SoC pipelines. We also conclude that the lockstep sequential consistency model provides well-defined ordering rules for lockstep execution and it can be implemented on an APU SoC without expensive hardware.

10.2 Future Research Directions

Here we list some future research directions to expand on the work presented in this dissertation.

10.2.1 Extending Spatiotemporal Coherence to Realtime Systems

Realt-time systems are more concerned about the predictability of cache coherence protocols than their performance [59]. The reason being a real-time schedule is always built on worst-case guarantees and a protocol that can provide a better worst-case guarantee is far better than a protocol that has better average performance.

The epochs in a Spatiotemporal Coherence acts as synchronization points for making the writes visible to the entire system. Thus, the time to grant all the epochs in a round-robin fashion represents the worst-case write completion latency. This predictability makes Spatiotemporal Coherence an attractive candidate for real-time coherence. From a predictable performance stand point, optimizations such as epoch skipping does not provide any advantage because the worst-case guarantee is based on the time to rotate all epochs like in the baseline Spatiotemporal Coherence implementation without any optimizations.

Prior work has demonstrated several types of coherence activity interferences that lead to unpredictability in memory access time [59]. Many of these interferences are caused by the coherence protocol trying to maintain a single-writer invariant. Since Spatiotemporal Coherence does not rely on single-writer invariant for enabling coherence, Spatiotemporal Coherence is immune to these issues. Thus, it is a natural fit for providing coherence for real-time systems.

10.2.2 Strong Models on Spatiotemporal Coherence

While we motivated the need for strong model on a GPU and proposed the low overhead Spatiotemporal Coherence with write atomicity support, we did not present an implementation of strong model on it. Many existing solutions to support strong ordering rely on the speculation capability of CPU cores [44, 45, 86] or on the underlying cache coherency mechanism [3, 87, 98, 100]. However, the GPU cores lack speculation support making the speculation based approaches unimplementable on a GPU. The cache coherency based approaches either rely on CPU style read-for-ownership (RFO) caches or on timestamp based

Program order	Epoch order #1	Epoch order #2	Epoch order #3
1. Store B	1. Epoch B	1. Epoch C	1. Epoch C
2. Load C	2. Epoch C	2. Epoch B	2. Epoch A
3. Load A	3. Epoch A	3. Epoch A	3. Epoch B
(a)	(b)	(c)	(d)

Figure 10.1: Interaction between epochs and mutexes. While mutexes protect conflicting accesses from reordering, epochs determine the modifiable addresses and hence can hint at potential access conflicts. Thus, by carefully orchestrating the epochs and creating a favorable epoch order, Spatiotemporal Coherence can reduce the mutex requirements for implementing a strong model.

coherence mechanisms. Spatiotemporal Coherence is neither RFO nor time-stamp based, making these implementations unattractive on it.

So, we have to rely on implementations that do not leverage any of the features mentioned earlier. Lin *et al.* [67] made the observation that strong ordering can be enforced by preventing reordering of conflicting accesses. We can leverage this observation for implementing strong models on Spatiotemporal Coherence. For preventing the reordering of conflicting accesses, one option is to leverage the mutex mechanism we adopted to implement Lockstep SC (Section 9.2. However, Spatiotemporal Coherence already has native support for detecting conflicts via epochs. Leveraging that support is possibly the best option for a future implementation.

Epoch-aware Mutex Mechanism

While mutexes protect conflicting accesses from reordering, epochs determine the modifiable addresses and hence can hint at potential access conflicts. So, only those accesses for which an epoch is granted need to be protected by mutexes. Figure 10.1(a) shows a program order where a store to epoch_B is followed by loads to $\text{epochs}_{C,A}$. The mutex acquisition rules demand both the loads to acquire mutexes if they are to be issued out of order before the older store's retirement [46]. The objective of this mutex acquisition is to prevent modification

of those load addresses before the retirement of the older store. However, Spatiotemporal Coherence guarantees that these loads will be modified only in epochs$_{C,A}$. Consequently, an epoch cycle where epoch$_B$ arrives before epochs$_{C,A}$ as shown in Figure 10.1(b) can completely eliminate the need to acquire mutexes for those speculative loads. Another epoch cycle as shown in Figure 10.1(c) will require only *Load C* to acquire mutex before the beginning of epoch$_C$. The only scenario where both loads need to acquire mutex is shown in Figure 10.1(d) when both epochs$_{C,A}$ occur before epoch$_B$, thus creating a possibility of an access conflict. Thus, by carefully orchestrating the epoch occurrence, Spatiotemporal Coherence's epoch rules can reduce the mutex requirements for implementing a strong model. It should be noted that mutex represents one of the conflict detection and ordering mechanisms but epochs can work in tandem with any other mechanism reducing the overhead associated with conflict detection.

10.2.3 Hierarchical Coherence Implementation

Extending Spatiotemporal Coherence hierarchically is straightforward. For example, suppose we are implementing hierarchical Spatiotemporal Coherence in a multi-GPU system. Each GPU can implement a level-0 Spatiotemporal Coherence with its own epoch management unit (EMU), and another instance of a level-1 Spatiotemporal Coherence managing inter-GPU coherence. The only difference here from a regular implementation is that the level-0 instances can only grant epochs already granted to it by the level-1 instance. The EMUs of the level-0 instances must be modified to enable this. Additionally, level-0 EMUs must be able to request epochs to the level-1's EMU. Other than these changes, the implementation should be the same as the implementation presented in this dissertation.

Such a hierarchical scheme must be able to provide coherence to a multi-GPU system or even to an APU system. However, in an APU system, the CPU coherence protocol must be the latency-sensitive Entangled Coherence. Below, we briefly discuss some potential modifications to Entangled Coherence to support CPU's write back cache and TSO consistency

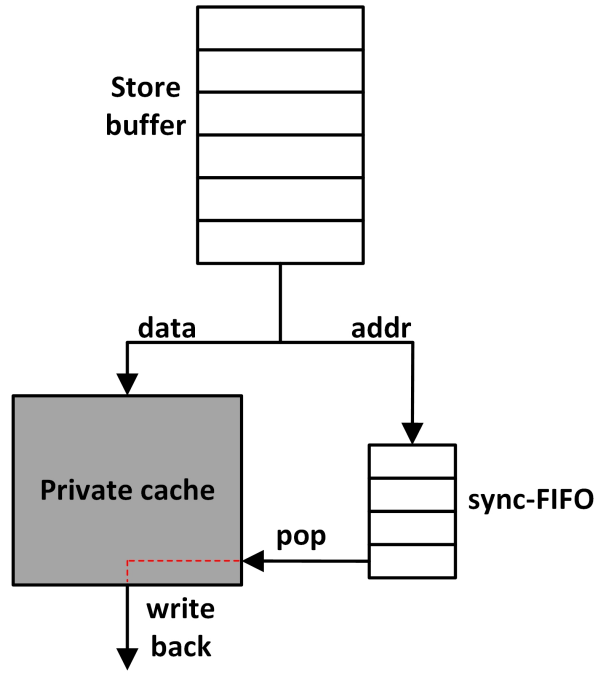


Figure 10.2: Private cache with synchronization FIFO. Synchronization FIFO maintains the program order of store and retires them in that order thus assisting Entangled Coherence to implement TSO consistency model.

model.

Entangled Coherence on CPU

Entangled Coherence as proposed in this dissertation (Section 4.2) has two major drawbacks when applied to a CPU: (a) the write-back property of CPU caches are unutilized, and (b) writes from the store buffer (SB) will be blocked from draining frequently waiting on an epoch. Both these drawbacks have the potential to stall the CPU pipeline due to non-availability of SB entries. We propose two optimizations to address these drawbacks: (a) extending SB to private caches, and (b) epoch aware eviction policy.

Extending Store Buffer to Private Caches: This optimization aims at increasing the effective size of SB by extending it to the private cache. This optimization also exploits the write back property of private caches. With this optimization, the stores are drained

from the store buffer but they are cached in the write-back cache. However, since TSO mandates that the stores from that processor should be made visible to peers in order, this implementation augments the private caches with a synchronization FIFO (sync-FIFO) as shown in Figure 10.2. Whenever a store is moved from the SB to the cache, the address of that store in pushed into the sync-FIFO. Thus, this sync-FIFO maintains the program order of stores from that processor. Such a sync-FIFO was proposed by Blake *et al.* to enable low overhead write combining for release consistency caches and to precisely flush stores on a release operation [49]. The primary purpose of sync-FIFO in that paper was to prevent full cache walks by storing the dirty addresses but in Entangled Coherence it has the additional responsibility of making stores visible outside this processor in order, which is not a requirement for release operation in the release-consistency model.

The writes are retired only in their corresponding epoch. So, at the end of each epoch, the sync-FIFO is consulted to see if the head entry in the FIFO corresponds to the current epoch. If the head entry corresponds to the current epoch, that entry is popped out of the FIFO and that dirty address is written back. This process is repeated till the FIFO is empty or the FIFO head is pointing to a store that does not belong to the epoch.

While this implementation increases the effective size of SB by extending it to the private cache, it imposes certain conditions for evicting a dirty cache line. For example, a dirty cache line chosen as the victim cannot be evicted if it is not at the sync-FIFO head. Even if it is at the head, it cannot be evicted if that address does not belong to the current epoch. This can result in the eviction getting blocked frequently, which in turn can block the demand request ultimately stalling the execution pipeline. Next we discuss our proposed optimization to address this eviction stalling challenge.

Epoch-aware Eviction Policy: One possible solution to this eviction stalling challenge is to drain all FIFO entries that are older than the victim entry and then allow the eviction of victim. However, this is possible only if the current epoch matches that of the older entries and of the victim entry. Since such a scenario can happen only rarely, this solution's

applicability is very limited. Instead, we propose epoch-aware evictions.

Epoch-aware eviction policy prioritizes victimization of the non-dirty addresses that does not belong to the current epoch over others. The valid entries of the current epoch will be anyway evicted by the lazy invalidation mechanism that was discussed earlier (Section 7.1.2). Thus, with epoch-aware eviction, the evictions will stall only if all entries in a cache set are dirty.

So far we discussed extending Spatiotemporal Coherence hierarchically. Next we will discuss hierarchical implementation of Spatiotemporal Coherence with a Non-spatiotemporal coherence protocol.

Hierarchical Coherence with a Non-spatiotemporal Coherence

There are two possibilities here: (a) level-0 is Spatiotemporal Coherence and level-1 is a non-spatiotemporal coherence implementation, and (b) vice-versa. Without loss of generality, we will discuss both these scenarios with an invalidation based protocol representing the non-spatiotemporal coherence implementation.

The implementation is straightforward with a level-0 Spatiotemporal Coherence protocol and level-1 invalidation-based protocol. Whenever an invalidation arrives at the EMU of level-0, EMU has to *forward the invalidation* to its private caches. This is trivially achieved by granting an epoch to the band corresponding to that invalidation address. This might incur significant performance penalty because an entire band of addresses is invalidated for an invalidation request to a single cache line. So, it may be better to build such a hierarchical coherence scheme with an invalidation protocol maintaining coherence at region granularity [80] with the address to band mapping matching the address to region mapping.

An implementation with level-1 Spatiotemporal Coherence and level-0 invalidation protocol is trickier. The invalidation based protocol must be augmented with a translation layer to convert write requests to epoch requests and to perform invalidations to all the addresses belonging to a band while granting an epoch to it. More research is needed to make this a

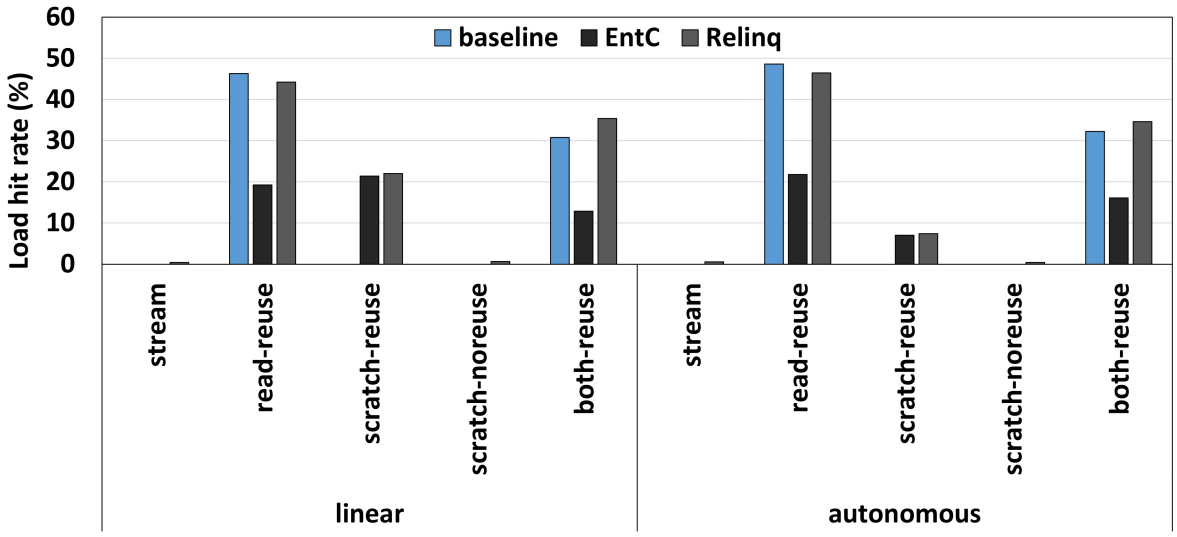


Figure 10.3: Efficiency of epoch relinquishing optimization. Epoch relinquishing optimization was able to bring the load hit rate back to the baseline level even for the variant with read-reuse that represented the degenerate case for Entangled Coherence (Section 8.3).

well-performing implementation.

10.2.4 Optimizations to Spatiotemporal and Entangled Coherence

While we proposed some performance optimizations to Spatiotemporal Coherence in Chapter 4, this novel coherence mechanism opens up a huge design space to explore and optimize. We will list some of them below:

1. Extending the implementation to logical time.

2. More sophisticated epoch to band mapping.

3. Sophisticated adaptive band formation techniques including extending the band formation techniques used by Entangled Coherence (Section 8.1.1) to Spatiotemporal Coherence.

4. Changing the epoch duration from a static time interval to a dynamically changing time interval.

5. Relinquishing an epoch immediately after writes are completed to a band: We identified this as the rootcause for the performance degradation observed with Entangled Coherence in Section 8.3.1. We implemented a simple version of this optimization to demonstrate its effectiveness for the benchmarks we evaluated in Section 8.3. The load hit rate improvement with this optimization is shown in Figure 10.3 (shown as *relinq*). This optimization was able to bring the load hit rate back to the baseline level even for the variant with read-reuse that represented the degenerate case for Entangled Coherence (Section 8.3).

6. Software driven optimizations: While Spatiotemporal Coherence does not require software assistance for its correct operation, it can take advantage of software driven optimizations such as software/programmer defined bands, software classification of read, write or read-write address regions for band formation, band-aware data allocation such that the allocation is contiguous in memory aiding in easy band formation, or even the memory allocator sending hints to the epoch manager about the efficient address to band mapping. If the band is exposed to the programmer, a programmer can also optimize a code's access pattern for band locality - that is consecutive accesses are issued to the same band - for additional performance benefits.

10.2.5 Speculative Execution

One of the most widely used approach for rolling back a speculative load execution is to forward an eviction notification to the speculating core notifying it to retry that load [42]. However, since Spatiotemporal Coherence does not cache accesses during an epoch, an eviction notification will never be generated and the loads belonging to the current epoch cannot be executed speculatively because the approach discussed above can no longer detect a spec-

ulation failure. This can result in performance loss. However, epochs also guarantee that no other bands are modified in that window of time, which means there is no need to forward eviction notification for the loads that do not belong to the current epoch. Thus, speculative execution presents both challenges and opportunities for Spatiotemporal Coherence, and may present a potential future research opportunity.

10.3 Reflections

In this section, I will share some of my general thoughts regarding the research presented in this dissertation and the future of heterogeneous SoC research in general. I will list them as questions and provide my thoughts on them so that a reader does not feel my opinions are impressed on them.

10.3.1 Are We Over-reliant on CPU Ideas?

We may be borrowing too many ideas from the CPU world and trying to apply it to other accelerators that do not share the same assumptions. For example, there is a general impression that implicit hardware-only cache coherence is not a viable option in a GPU. However, we arrived at this conclusion by evaluating a CPU-style invalidation based protocol which was designed for multi-core CPUs. Another example of this would be extending sequential consistency to GPUs without considering the ordering implications of lockstep executing threads.

I would like to propose a different way of thinking about the GPU challenges. Let us suppose GPUs were invented first and CPUs came out later. The architecture community still would have invented solutions to these problems but I believe they would look very different from the CPU inspired solutions that we are re-purposing today. An example of this would be Spatiotemporal Coherence itself and its shift away from processor-centric write-permission. This unlearning of CPU ideas may be especially important with the heterogeneous SoC space

now inundated with different types of processing elements with varying characteristics [52] and all of them warrant ingenious approaches tailored for them. However, this is not saying we should completely shun the ideas from the CPU world but be more selective in applying them to the new space.

10.3.2 What Should be the Simulation Methodology?

A heterogeneous SoC has different type of processing elements and modeling all of them is next to impossible. Same goes with writing benchmarks for all the diverse use cases that can target an heterogeneous SoC. Hill and Reddi addressed this issue in their work [52] and came up with an analytical model. We would like to propose a different approach. Our proposal is to use a generic traffic generator application like the one we used for evaluating Entangled Coherence (Section 6.3.3). Every software stage can be emulated reasonably accurately with a fixed set of parameterizable features. While this approach might not have cycle level accuracy, it can be useful for early design space exploration stage.

Mirroring the same approach on the simulator side, instead of using a cycle level model of a processing engine, we propose using a simple model that can satisfy the compute to bandwidth ratio, and generate and issue requests to the memory at the rate demanded by the software stage running on it. Such an approach will save the time of both academic and industry researchers working on early stage design exploration and prototyping.

The idea can be even implemented on a prototype SoC instead of a simulator. The major advantage of a simulator compared to a prototype SoC is the flexibility it offers with regards to changing the memory system features such as the cache coherence protocol. Even with these constraints, a prototype SoC still has value. For example, such a prototype can be used for understanding the data sharing characterestics of applications and guide the hardware design. The cache coherence protocol of that prototype SoC can be a generic protocol like Spatiotemporal Coherence or even Entangled Coherence that can provide functionally correct execution under all scenarios if not performing well.

Bibliography

[1] S. V. Adve and M. D. Hill. "Weak ordering-a new definition". In: *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*. May 1990, pp. 2–14. DOI: 10.1109/ISCA.1990.134502.

[2] Sarita V. Adve and Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial". In: *Computer* 29.12 (Dec. 1996), pp. 66–76. ISSN: 0018-9162. DOI: 10.1109/2.546611. URL: http://dx.doi.org/10.1109/2.546611.

[3] Sarita V. Adve and Mark D. Hill. "Implementing Sequential Consistency in Cache-Based Systems". In: *In Proceedings of the 1990 International Conference on Parallel Processing*. 1990, pp. 47–50.

[4] Sarita Vikram Adve. "Designing Memory Consistency Models for Shared-memory Multiprocessors". UMI Order No. GAX94-07354. PhD thesis. Madison, WI, USA, 1993.

[5] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. "GPU Concurrency: Weak Behaviours and Programming Assumptions". In: *SIGPLAN Not.* 50.4 (Mar. 2015), pp. 577–591. ISSN: 0362-1340. DOI: 10.1145/2775054.2694391. URL: http://doi.acm.org/10.1145/2775054.2694391.

[6] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. "Software Verification for Weak Memory via Program Transformation". In: *CoRR* abs/1207.7264 (2012). arXiv: 1207.7264. URL: http://arxiv.org/abs/1207.7264.

[7] Jade Alglave and Luc Maranget. "Stability in Weak Memory Models". In: *CAV*. 2011.

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014), 7:1–7:74. ISSN: 0164-0925. DOI: 10.1145/2627752. URL: http://doi.acm.org/10.1145/2627752.

[9] M. Alisafaee. "Spatiotemporal Coherence Tracking". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 341–350.

[10] J. Alsop, M. Sinclair, and S. Adve. "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 261–274. DOI: 10.1109/ISCA.2018.00031.

[11] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. "Lazy Release Consistency for GPUs". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 26:1–26:13. URL: http://dl.acm.org/citation.cfm?id=3195638.3195669.

[12] AMD. *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. 2012. URL: https://goo.gl/GPvy8R.

[13] AMD. *AMD GCN3 ISA Architecture Manual*. 2016. URL: https://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual.

[14] AMD. *User Guide for AMDGPU Backend*. 2019. URL: https://llvm.org/docs/AMDGPUUsage.html.

[15] AMD. *Compute Apps*. URL: https://github.com/AMDComputeLibraries/ComputeApps.

[16] AMD. *HC*. URL: https://rocm.github.io/languages.html.

[17] AMD. *HCC*. URL: https://github.com/RadeonOpenCompute/hcc/wiki.

[18] AMD. *HCC Example Apps*. URL: https://github.com/ROCm-Developer-Tools/HCC-Example-Application.

[19] AMD. *HIP*. URL: https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code.

[20] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. URL: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[21] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. "On the Verification Problem for Weak Memory Models". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 7–18. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706303. URL: http://doi.acm.org/10.1145/1706299.1706303.

[22] Arkaprava Basu, Sooraj Puthoor, Shuai Che, and Bradford M. Beckmann. "Software Assisted Hardware Cache Coherence for Heterogeneous Processors". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: ACM, 2016, pp. 279–288. ISBN: 978-1-4503-4305-3. DOI: 10.1145/2989081.2989092. URL: http://doi.acm.org/10.1145/2989081.2989092.

[23] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. "Juggler: A Dependence-aware Task-based Execution Framework for GPUs". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: ACM, 2018, pp. 54–67. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178492. URL: http://doi.acm.org/10.1145/3178487.3178492.

[24] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. "InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 233–244. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555785. URL: http://doi.acm.org/10.1145/1555754.1555785.

[25] 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 '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 68–78. ISBN: 9781595938602. DOI: 10.1145/1375581.1375591. URL: https://doi.org/10.1145/1375581.1375591.

[26] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. "DAGuE: A Generic Distributed DAG Engine for High Performance Computing". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011, pp. 1151–1158. DOI: 10.1109/IPDPS.2011.281.

[27] D. Bouvier and B. Sander. "Applying AMD's Kaveri APU for heterogeneous computing". In: *2014 IEEE Hot Chips 26 Symposium (HCS)*. 2014.

[28] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. "CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 12–21. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250737. URL: http://doi.acm.org/10.1145/1250734.1250737.

[29] Sebastian Burckhardt and Madanlal Musuvathi. "Effective Program Verification for Relaxed Memory Models". In: *Proceedings of the 20th International Conference on Computer Aided Verification*. CAV '08. Princeton, NJ, USA: Springer-Verlag, 2008,

pp. 107–120. ISBN: 978-3-540-70543-7. DOI: 10.1007/978-3-540-70545-1_12. URL: http://dx.doi.org/10.1007/978-3-540-70545-1_12.

[30] Jacob Burnim, Koushik Sen, and Christos Stergiou. "Testing Concurrent Programs on Relaxed Memory Models". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 122–132. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001436. URL: http://doi.acm.org/10.1145/2001420.2001436.

[31] H. W. Cain and M. H. Lipasti. "Memory ordering: a value-based approach". In: *IEEE Micro* 24.6 (Nov. 2004), pp. 110–117. ISSN: 0272-1732. DOI: 10.1109/MM.2004.81.

[32] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. "BulkSC: Bulk Enforcement of Sequential Consistency". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 278–289. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250697. URL: http://doi.acm.org/10.1145/1250662.1250697.

[33] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. "Pannotia: Understanding irregular GPGPU graph applications". In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 2013.

[34] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[35] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism". In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011, pp. 155–166.

[36] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 608–621. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837615. URL: http://doi.acm.org/10.1145/2837614.2837615.

[37] HSA Foundation. *HSA Platform System Architecture Specification 1.1*. 2016. URL: http://www.hsafoundation.com/?ddownload=5114.

[38] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. Dec. 2007, pp. 407–420. DOI: 10.1109/MICRO.2007.30.

[39] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. "Hardware transactional memory for GPU architectures". In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2011, pp. 296–307.

[40] Benedict R Gaster. "HSA memory model." In: *Hot Chips Symposium*. 2013, pp. 1–42.

[41] Benedict R. Gaster, Derek Hower, and Lee Howes. "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models". In: *ACM Trans. Archit. Code Optim.* 12.1 (Apr. 2015). ISSN: 1544-3566. DOI: 10.1145/2701618. URL: https://doi.org/10.1145/2701618.

[42] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. "Two Techniques to Enhance the Performance of Memory Consistency Models". In: *In Proceedings of the 1991 International Conference on Parallel Processing*. 1991, pp. 355–364.

[43] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors". In: *Proceedings of the 17th Annual International Sympo-*

sium on Computer Architecture. ISCA '90. Seattle, Washington, USA: ACM, 1990, pp. 15–26. ISBN: 0-89791-366-3. DOI: 10.1145/325164.325102. URL: http://doi.acm.org/10.1145/325164.325102.

[44] C. Gniady and B. Falsafi. "Speculative sequential consistency with little custom storage". In: *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. 2002, pp. 179–188. DOI: 10.1109/PACT.2002.1106016.

[45] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. "Is SC + ILP = RC?" In: *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ISCA '99. Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 162–171. ISBN: 0-7695-0170-2. DOI: 10.1145/300979.300993. URL: http://dx.doi.org/10.1145/300979.300993.

[46] D. Gope and M. H. Lipasti. "Atomic SC for simple in-order processors". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 404–415. DOI: 10.1109/HPCA.2014.6835950.

[47] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 608–619. DOI: 10.1109/HPCA.2018.00058.

[48] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 184–195. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555779. URL: http://doi.acm.org/10.1145/1555754.1555779.

[49] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. "QuickRelease: A throughput-oriented approach to release consistency on GPUs". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 189–200. DOI: 10.1109/HPCA.2014.6835930.

[50] Blake A. Hechtman and Daniel J. Sorin. "Exploring Memory Consistency for Massively-threaded Throughput-oriented Processors". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 201–212. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485940. URL: http://doi.acm.org/10.1145/2485922.2485940.

[51] Stephan Heinrich and Lucid Motors. "Flash memory in the emerging age of autonomy". In: *Flash Memory Summit* (2017), pp. 1–10.

[52] Mark Hill and Vijay Janapa Reddi. "Gables: A Roofline Model for Mobile SoCs". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 317–330. DOI: 10.1109/HPCA.2019.00047.

[53] Mark D. Hill. "Multiprocessors Should Support Simple Memory-Consistency Models". In: *Computer* 31.8 (Aug. 1998), pp. 28–34. ISSN: 0018-9162. DOI: 10.1109/2.707614. URL: http://dx.doi.org/10.1109/2.707614.

[54] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "Heterogeneous-race-free Memory Models". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 427–440. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541981. URL: http://doi.acm.org/10.1145/2541940.2541981.

[55] Mohamed Assem Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. "Analyzing and Leveraging Shared L1 Caches in GPUs". In: *Proceedings of*

the ACM International Conference on Parallel Architectures and Compilation Techniques. PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 161–173. ISBN: 9781450380751. DOI: 10.1145/3410463.3414623. URL: https://doi.org/10.1145/3410463.3414623.

[56] *Inside Volta: The World's Most Advanced Data Center GPU*. 2017. URL: https://devblogs.nvidia.com/inside-volta/.

[57] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. "A scalable architecture for ordered parallelism". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 228–241. DOI: 10.1145/2830772.2830777.

[58] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. "MRPB: Memory request prioritization for massively parallel processors". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 272–283. DOI: 10.1109/HPCA.2014.6835938.

[59] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. "Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems". In: *IEEE Transactions on Computers* 70.12 (2021), pp. 2098–2111. DOI: 10.1109/TC.2020.3037747.

[60] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. "Efficient Utilization of GPGPU Cache Hierarchy". In: *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. GPGPU-8. San Francisco, CA, USA: ACM, 2015, pp. 36–47. ISBN: 978-1-4503-3407-5. DOI: 10.1145/2716282.2716291. URL: http://doi.acm.org/10.1145/2716282.2716291.

[61] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. "Meet the Walkers: Accelerating Index Traversals for In-memory Databases". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 468–479. ISBN: 978-1-4503-

2638-4. DOI: 10.1145/2540708.2540748. URL: http://doi.acm.org/10.1145/2540708.2540748.

[62] G. Krishnan, D. Bouvier, and S. Naffziger. "Energy-Efficient Graphics and Multimedia in 28-nm Carrizo Accelerated Processing Unit". In: *IEEE Micro* (2016).

[63] Sohan Lal, Bogaraju Sharatchandra Varma, and Ben Juurlink. "A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads". In: *International Journal of Parallel Programming* 50.2 (2022), pp. 189–216.

[64] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. "ComP-net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: ACM, 2018, 29:1–29:13. ISBN: 978-1-4503-5986-3. DOI: 10.1145/3243176.3243179. URL: http://doi.acm.org/10.1145/3243176.3243179.

[65] 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.

[66] C. Lin, V. Nagarajan, and R. Gupta. "Efficient sequential consistency using conditional fences". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2010, pp. 295–306.

[67] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. "Efficient Sequential Consistency via Conflict Ordering". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 273–286. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2151006. URL: http://doi.acm.org/10.1145/2150976.2151006.

[68] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. "The Architectural Implications of Autonomous Driving: Constraints and Acceleration". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 751–766. ISBN: 9781450349116. DOI: 10.1145/3173162.3173191. URL: https://doi.org/10.1145/3173162.3173191.

[69] Yuan Lin and Vinod Grover. *Using CUDA Warp-Level Primitives*. 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[70] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 257–270. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304043. URL: http://doi.acm.org/10.1145/3297858.3304043.

[71] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. "ArMOR: Defending against Memory Consistency Model Mismatches in Heterogeneous Architectures". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: Association for Computing Machinery, 2015, pp. 388–400. ISBN: 9781450334020. DOI: 10.1145/2749469.2750378. URL: https://doi.org/10.1145/2749469.2750378.

[72] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. "An Axiomatic Memory Model for POWER Multiprocessors". In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berkeley, CA:

Springer-Verlag, 2012, pp. 495–512. ISBN: 978-3-642-31423-0. DOI: 10.1007/978-3-642-31424-7_36. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_36.

[73] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset". In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 92–99. ISSN: 0163-5964. DOI: 10.1145/1105734.1105747. URL: http://doi.acm.org/10.1145/1105734.1105747.

[74] Microsoft. *C++ AMP : Language and Programming Model*. URL: http://download.microsoft.com/download/2/2/9/22972859-15c2-4d96-97ae-93344241d56c/cppampopenspecificationv12.pdf.

[75] Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture". In: *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*. ICPP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 739–748. ISBN: 978-1-4673-7587-0. DOI: 10.1109/ICPP.2015.83. URL: http://dx.doi.org/10.1109/ICPP.2015.83.

[76] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. "An Analysis of Persistent Memory Use with WHISPER". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 135–148. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037730. URL: http://doi.acm.org/10.1145/3037697.3037730.

[77] NVIDIA. "NVIDIA TESLA V100 GPU ARCHITECTURE". In: (2009). URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[78] NVIDIA. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[79] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. "Fine-grain Task Aggregation and Coordination on GPUs". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 181–192. ISSN: 0163-5964. DOI: 10.1145/2678373.2665701. URL: http://doi.acm.org/10.1145/2678373.2665701.

[80] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "Heterogeneous System Coherence for Integrated CPU-GPU Systems". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 457–467. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540747. URL: http://doi.acm.org/10.1145/2540708.2540747.

[81] Jason Power, Mark D. Hill, and David A. Wood. "Supporting x86-64 address translation for 100s of GPU lanes". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 568–578. DOI: 10.1109/HPCA.2014.6835965.

[82] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. "Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 19:1–19:29. ISSN: 2475-1421. DOI: 10.1145/3158107. URL: http://doi.acm.org/10.1145/3158107.

[83] Sooraj Puthoor, Ashwin M. Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M. Beckmann, and Gregory Rodgers. "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture". In: *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU '16. Barcelona,

Spain: ACM, 2016, pp. 53–62. ISBN: 978-1-4503-4195-0. DOI: 10.1145/2884045.2884052. URL: http://doi.acm.org/10.1145/2884045.2884052.

[84] Sooraj Puthoor and Mikko H. Lipasti. "Compiler Assisted Coalescing". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243203. URL: https://doi.org/10.1145/3243176.3243203.

[85] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. "Oversubscribed Command Queues in GPUs". In: *Proceedings of the 11th Workshop on General Purpose GPUs*. GPGPU-11. Vienna, Austria: ACM, 2018, pp. 50–60. ISBN: 978-1-4503-5647-3. DOI: 10.1145/3180270.3180271. URL: http://doi.acm.org/10.1145/3180270.3180271.

[86] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models". In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '97. Newport, Rhode Island, USA: ACM, 1997, pp. 199–210. ISBN: 0-89791-890-8. DOI: 10.1145/258492.258512. URL: http://doi.acm.org/10.1145/258492.258512.

[87] X. Ren and M. Lis. "Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 625–636. DOI: 10.1109/HPCA.2017.40.

[88] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans. "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 582–595.

[89] A. Ros and S. Kaxiras. "Racer: TSO consistency via race detection". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783736.

[90] A. Ros and S. Kaxiras. "Non-Speculative Store Coalescing in Total Store Order". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 221–234. DOI: 10.1109/ISCA.2018.00028.

[91] Ben Sander. *AMD GCN ASSEMBLY: CROSS-LANE OPERATIONS*. 2016. URL: https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/.

[92] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER Multiprocessors". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 175–186. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993520. URL: http://doi.acm.org/10.1145/1993498.1993520.

[93] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. "Library Cache Coherence". In: (2011).

[94] M. D. Sinclair, J. Alsop, and S. V. Adve. "Efficient GPU synchronization without scopes: Saying no to complex consistency models". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2015, pp. 647–659. DOI: 10.1145/2830772.2830821.

[95] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 161–174. ISBN: 9781450348928. DOI: 10.1145/3079856.3080206. URL: https://doi.org/10.1145/3079856.3080206.

[96] A. Singh, S. Aga, and S. Narayanasamy. "Efficiently enforcing strong memory ordering in GPUs". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2015, pp. 699–712. DOI: 10.1145/2830772.2830778.

[97] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. "End-to-end Sequential Consistency". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 524–535. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337220.

[98] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. "Cache coherence for GPU architectures". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 578–590. DOI: 10.1109/HPCA.2013.6522351.

[99] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. ISBN: 1608455645, 9781608455645.

[100] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. "G-TSC: Timestamp Based Coherence for GPUs". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 403–415.

[101] 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*. 2015.

[102] D. Vantrease, M. H. Lipasti, and N. Binkert. "Atomic Coherence: Leveraging nanophotonics to build race-free cache coherence protocols". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011, pp. 132–143.

[103] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. "Mechanisms for Store-wait-free Multiprocessors". In: *SIGARCH Comput. Archit.*

News 35.2 (June 2007), pp. 266–277. ISSN: 0163-5964. DOI: 10.1145/1273440.1250696. URL: http://doi.acm.org/10.1145/1273440.1250696.

[104] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. "Hierarchical DAG Scheduling for Hybrid Distributed Systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 156–165. DOI: 10.1109/IPDPS.2015.56.

[105] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems". In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-51. Fukuoka, Japan: IEEE Press, 2018, pp. 339–351. ISBN: 9781538662403. DOI: 10.1109/MICRO.2018.00035. URL: https://doi.org/10.1109/MICRO.2018.00035.

[106] Xiangyao Yu and Srinivas Devadas. "Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory". In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 227–240. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.12. URL: https://doi.org/10.1109/PACT.2015.12.

[107] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. "Tardis 2.0: Optimized time traveling coherence for relaxed consistency models". In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE. 2016, pp. 261–274.

[108] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. "A Tagless Coherence Directory". In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 423–434. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669166. URL: http://doi.acm.org/10.1145/1669112.1669166.

[109] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. "Taming Weak Memory Models". In: *CoRR* abs/1606.05416 (2016). arXiv: 1606.05416. URL: http://arxiv.org/abs/1606.05416.

[110] Sizhuo Zhang, Muralidaran Vijayaraghavan, Andrew Wright, Mehdi Alipour, and Arvind. "Constructing a Weak Memory Model". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 124–137. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00021. URL: https://doi.org/10.1109/ISCA.2018.00021.