Memory Consistency Directed Cache Coherence Protocols for Scalable Multiprocessors

Marco Elver

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2016

Abstract

The memory consistency model, which formally specifies the behavior of the memory system, is used by programmers to reason about parallel programs. From a hardware design perspective, weaker consistency models permit various optimizations in a multiprocessor system: this thesis focuses on designing and optimizing the cache coherence protocol for a given target memory consistency model.

Traditional directory coherence protocols are designed to be compatible with the strictest memory consistency model, sequential consistency (SC). When they are used for chip multiprocessors (CMPs) that provide more relaxed memory consistency models, such protocols turn out to be unnecessarily strict. Usually, this comes at the cost of scalability, in terms of per-core storage due to sharer tracking, which poses a problem with increasing number of cores in today's CMPs, most of which no longer are sequentially consistent. The recent convergence towards programming language based relaxed memory consistency models has sparked renewed interest in lazy cache coherence protocols. These protocols exploit synchronization information by enforcing coherence only at synchronization boundaries via self-invalidation. As a result, such protocols do not require sharer tracking which benefits scalability. On the downside, such protocols are only readily applicable to a restricted set of consistency models, such as Release Consistency (RC), which expose synchronization information explicitly. In particular, existing architectures with stricter consistency models (such as x86) cannot readily make use of lazy coherence protocols without either: adapting the protocol to satisfy the stricter consistency model; or changing the architecture's consistency model to (a variant of) RC, typically at the expense of backward compatibility. The first part of this thesis explores both these options, with a focus on a practical approach satisfying backward compatibility.

Because of the wide adoption of Total Store Order (TSO) and its variants in x86 and SPARC processors, and existing parallel programs written for these architectures, we first propose TSO-CC, a lazy cache coherence protocol for the TSO memory consistency model. TSO-CC does not track sharers and instead relies on self-invalidation and detection of potential acquires (in the absence of explicit synchronization) using per cache line timestamps to efficiently and lazily satisfy the TSO memory consistency model. Our results show that TSO-CC achieves, on average, performance comparable to a MESI directory protocol, while TSO-CC's storage overhead per cache line scales logarithmically with increasing core count.

Next, we propose an approach for the x86-64 architecture, which is a compromise between retaining the original consistency model and using a more storage efficient lazy coherence protocol. First, we propose a mechanism to convey synchronization information via a simple ISA extension, while retaining backward compatibility with legacy codes and older microarchitectures. Second, we propose RC3 (based on TSO-CC), a scalable cache coherence protocol for RCtso, the resulting memory consistency model. RC3 does not track sharers and relies on self-invalidation on acquires. To satisfy RCtso efficiently, the protocol reduces self-invalidations transitively using per-L1 timestamps only. RC3 outperforms a conventional lazy RC protocol by 12%, achieving performance comparable to a MESI directory protocol for RC optimized programs. RC3's storage overhead per cache line scales logarithmically with increasing core count and reduces on-chip coherence storage overheads by 45% compared to TSO-CC.

Finally, it is imperative that hardware adheres to the promised memory consistency model. Indeed, consistency directed coherence protocols cannot use conventional coherence definitions (e.g. SWMR) to be verified against, and few existing verification methodologies apply. Furthermore, as the full consistency model is used as a specification, their interaction with other components (e.g. pipeline) of a system must not be neglected in the verification process. Therefore, verifying a system with such protocols in the context of interacting components is even more important than before. One common way to do this is via executing tests, where specific threads of instruction sequences are generated and their executions are checked for adherence to the consistency model. It would be extremely beneficial to execute such tests under simulation, i.e. when the functional design implementation of the hardware is being prototyped. Most prior verification methodologies, however, target post-silicon environments, which when used for simulation-based memory consistency verification would be too slow.

We propose McVerSi, a test generation framework for fast memory consistency verification of a full-system design implementation under simulation. Our primary contribution is a Genetic Programming (GP) based approach to memory consistency test generation, which relies on a novel crossover function that prioritizes memory operations contributing to non-determinism, thereby increasing the probability of uncovering memory consistency bugs. To guide tests towards exercising as much logic as possible, the simulator's reported coverage is used as the fitness function. Furthermore, we increase test throughput by making the test workload simulation-aware. We evaluate our proposed framework using the Gem5 cycle accurate simulator in full-system mode with Ruby (with configurations that use Gem5's MESI protocol, and our proposed

TSO-CC together with an out-of-order pipeline). We discover 2 new bugs in the MESI protocol due to the faulty interaction of the pipeline and the cache coherence protocol, highlighting that even conventional protocols should be verified rigorously in the context of a full-system. Crucially, these bugs would not have been discovered through individual verification of the pipeline or the coherence protocol. We study 11 bugs in total. Our GP-based test generation approach finds all bugs consistently, therefore providing much higher guarantees compared to alternative approaches (pseudo-random test generation and litmus tests).

Lay Summary

At the beginning of the 21st century, designers of high-performance processors found that increasing performance by increasing clock frequencies become infeasible. Further increasing clock frequencies come with unmanageable energy budgets as well as complexities in the way instructions would be processed. Yet, Moore's law is indeed still going strong, and we are seeing more and more transistors on a single chip. Consequently, this potential should not be ignored, and designers turned towards replicating the processing "cores" responsible for a single "thread" of instructions on a single chip: the result is the "multicore" chip, which can process many more threads at the same time, therefore improving overall performance.

Unfortunately, with the current designs, problems arise when attempting to scale to an order of magnitude more cores than we have today. Current state-of-the-art approaches for specific components used on a chip are simply not scalable in terms of the number of cores. One such component is the "cache coherence protocol." These days, every processing core has attached to it a small private cache to speed up memory accesses. These caches replicate data that is found in main memory. However, if two cores access the same data, then this data would be replicated in these two private caches. Now, what happens if one of these cores modifies this data and the other tries to read it? It is the coherence protocol's job to ensure there are no inconsistencies in the caches by performing various operations (sending invalidations, updates, new data, etc.).

From the programmer's point of view, caches should be transparent. The programmer, however, sticks to certain rules which specify what happens when multiple threads read and write the same data: these rules are called the "memory consistency model." The memory consistency model simply says in what order operations must become visible relative to each other. The processor must then ensure that these rules are not broken to guarantee consistency. In the mentioned case where the cores each have private caches, and the same data is replicated across them, it may suddenly be possible to violate these rules if the cache coherence protocol did not exist.

Traditional cache coherence protocols, however, do not scale well to many more cores. The reason for this is that they require maintaining metadata, which will take up too much space on a chip if more cores are added. To solve this problem, there have been several recent works which propose new cache coherence protocols which do not have this problem by eliminating the expensive metadata. The commonality among

them is, that they rely on relaxed memory consistency models (those that enforce very little ordering among instructions). Unfortunately, these proposals rely on consistency models which either are not widely used or not used at all in real processors. This limits their applicability to modern architectures used in today's and future devices, as too much existing software is too important and cannot simply stop working.

In this thesis, this new class of coherence protocol is explored in the context of real architectures, in particular, the common x86 and SPARC architectures. The consistency model found these systems is called Total Store Order (TSO). The first contribution of this thesis is called TSO-CC, a coherence protocol for TSO, which does not require the use of expensive metadata and achieves performance comparable to a traditional coherence protocol. TSO-CC achieves this via a novel technique using timestamps to limit the times when the protocol needs to perform costly maintenance on private caches.

The second contribution is RC3, a coherence protocol optimized for a more relaxed consistency model than TSO, as these types of consistency models are found in modern programming languages—ideally, the consistency models of multicores and programming languages should be aligned for optimal performance. Unlike previous approaches, the presented approach is still compatible with the x86 architecture and achieves good performance by reusing the timestamp technique—but in a limited form—from TSO-CC. Consequently, the RC3 protocol is even more efficient than TSO-CC.

Many of these techniques are usually prototyped as part of a processor simulator. In particular, the simulation of an entire computer is called a "full-system simulation." Here, the coherence protocol is only a small component of the larger system but interacts with many other components. These other components may also have an effect on the enforcement of the promised memory consistency model. In particular, some interactions between coherence protocol and processing pipeline are crucial. For the "consistency directed coherence protocols" this thesis proposes, these interactions should not be neglected; however, we found even traditional protocols are not checked sufficiently for the interaction with other components. To ensure the processor does not violate the consistency model's rules, the third and final contribution, called McVerSi, is an approach for fast verification of a full-system implemented in a processor simulator. Indeed, there is a gap in existing approaches, as they do not optimize (they are too slow) for memory consistency verification in a full-system simulation. McVerSi is filling this gap.

Acknowledgements

First and foremost, I wish to thank my advisor, Dr. Vijayanand Nagarajan, for his guidance, support and patience throughout. His advice was invaluable in helping me understanding the topics at hand and the process of research itself. Furthermore, I am extremely thankful for Vijay continually encouraging me to aim higher, as well as giving me the space to explore.

Second, I would like to thank my second supervisor, Christian Fensch. I am very thankful for his honest and critical advice, which helped me see things from different perspectives. I would also like to thank Björn Franke and Susmit Sarkar for being part of the annual review committee and their helpful suggestions. Many thanks to Babak Falsafi and Boris Grot for being on my viva panel.

On this journey, many people have supported, given advice, inspired my curiosity and/or helped in various ways—thank you! Special mention goes to: Andrew McPherson, Andrew Sogokon, Arpit Joshi, Bharghava Rajaram, Cheng-Chieh Huang, Chris Banks, Chris Margiolas, Christian Buck, Christopher Thompson, Erik Tomusk, Gabriele Farina, Harry Wagstaff, Jade Alglave, Jean-Luc Stevens, Ohad Kammar, Paul Jackson, Peter Sewell, Philipp Rüdiger, Saumay Dublish, Stephan Diestelhorst, Thibaut Lutz, Vasileios Porpodas, and everybody at ICSA.

Last, but not least, my sincere thanks to my parents for their continued support and encouragement.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

Marco Elver and Vijay Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO", in *20th IEEE International Symposium on High Performance Computer Architecture (HPCA-20)*, Orlando, FL, USA, February 15-19, 2014. DOI: 10.1109/HPCA.2014.6835927

Marco Elver and Vijay Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions", in *International Conference on Parallel Architectures and Compilation Techniques (PACT '15)*, San Francisco, CA, USA, October 18-21, 2015. DOI: 10.1109/PACT.2015.37

Marco Elver and Vijay Nagarajan, "McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation", in *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA-22)*, Barcelona, Spain, March 12-16, 2016. DOI: 10.1109/HPCA.2016.7446099

(*Marco Elver*)

To my parents.

Table of Contents

I Preamble **1**

1 Introduction **3**

1.1 Cache Coherence Protocol Scaling 3

1.2 Memory Consistency and Cache Coherence 6

1.3 Contributions 7

1.4 Thesis Structure 9

II Background **11**

2 Memory Consistency Models **13**

2.1 Overview 13

2.2 Axiomatic Framework 14

2.2.1 Instruction Semantics 14

2.2.2 Candidate Executions 15

2.2.3 Architecture Definitions 17

2.2.4 Constraint Specifications 17

2.3 Assumptions on Progress Guarantees 18

2.4 System-Centric Models 19

2.4.1 Sequential Consistency 19

2.4.2 Total Store Order 19

2.4.3 Release Consistency 21

2.5 Programmer-Centric Models 21

3 Cache Coherence Protocols **25**

3.1 Overview 25

3.2 Definition of Coherence 25

3.3 Baseline and Assumptions 28

3.3.1 Adding the Exclusive State 31
3.4 Eager versus Lazy Coherence 31

III Consistency Directed Cache Coherence Protocols 35

4 TSO-CC: Consistency Directed Cache Coherence for TSO 37
4.1 Introduction 37
4.1.1 Motivation 38
4.1.2 Requirements 38
4.1.3 Approach 39
4.2 TSO-CC: Protocol Design 40
4.2.1 Overview 40
4.2.2 Basic Protocol 42
4.2.3 Opt. 1: Reducing Self-Invalidations 43
4.2.4 Opt. 2: Shared Read-Only Data 44
4.2.5 Timestamp Resets 46
4.2.6 Atomic Accesses and Fences 47
4.2.7 Speculative Execution 47
4.2.8 Storage Requirements and Organization 48
4.3 Proof of Correctness 50
4.3.1 Abstract TSO Load-Buffering Machine 51
4.3.2 Sketch for Unoptimized Protocol 55
4.4 Evaluation Methodology 57
4.4.1 Simulation Environment 57
4.4.2 Workloads 57
4.4.3 Protocol Configurations and Storage Overheads 58
4.4.4 Verification 61
4.5 Experimental Results 62
4.5.1 Discussion 67
4.6 Related Work 68
4.6.1 Coherence for Sequential Consistency 68
4.6.2 Coherence for Relaxed Consistency Models 69
4.6.3 Distributed Shared Memory (DSM) 70
4.7 Conclusion 70

5 RC3: Consistency Directed Cache Coherence for x86-64 with RC Extensions 73

5.1 Introduction . . . 73
5.1.1 Motivation . . . 74
5.1.2 Approach . . . 74
5.2 Limitations of TSO-CC . . . 75
5.3 x86-RCtso: Release Consistency for x86-64 . . . 77
5.3.1 ISA Extension Details . . . 78
5.4 RC3: Protocol Design . . . 79
5.4.1 Overview . . . 79
5.4.2 Basic Protocol . . . 80
5.4.3 Opt. 1: Reducing Self-Invalidations of Redundant Acquires . . . 82
5.4.4 Timestamp Resets . . . 84
5.4.5 Opt. 2: Shared Read-Only with Epoch Based Decay . . . 85
5.4.6 Atomic Instructions and Fences . . . 86
5.4.7 Speculative Execution . . . 86
5.4.8 Storage Requirements and Organization . . . 86
5.5 Evaluation Methodology . . . 88
5.5.1 Simulation Environment . . . 88
5.5.2 Workloads . . . 89
5.5.3 Protocol Configurations and Storage Overheads . . . 90
5.6 Experimental Results . . . 91
5.7 Related Work . . . 96
5.7.1 Language to Hardware Level Consistency . . . 96
5.7.2 Consistency Directed Coherence . . . 97
5.7.3 Data Structures in Eager Protocols . . . 97
5.8 Conclusion . . . 98

IV Memory Consistency Verification 101

6 McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation 103

6.1 Introduction . . . 103
6.1.1 Approach . . . 104
6.2 Evolutionary Algorithms . . . 106

6.3 Test Generation . . . 106
6.3.1 Overview . . . 107
6.3.2 Coverage and Fitness . . . 109
6.3.3 Test Representation, Crossover and Mutation . . . 110
6.4 Accelerating Test Execution & Checking . . . 114
6.4.1 Checker . . . 114
6.4.2 Complexity Implications . . . 116
6.5 Evaluation Methodology . . . 117
6.5.1 Simulation Environment . . . 117
6.5.2 Test Generation & Checking . . . 118
6.5.3 Selected Bugs . . . 121
6.6 Experimental Results . . . 123
6.6.1 Bug Coverage . . . 123
6.6.2 Structural Coverage . . . 126
6.7 Related Work . . . 127
6.7.1 Formal Verification . . . 127
6.7.2 Memory System Verification . . . 128
6.7.3 Full-System Verification . . . 128
6.7.4 Hardware Support for MCM Verification . . . 129
6.8 Conclusion . . . 129

V Conclusions 131

7 Conclusions and Future Directions 133
7.1 Opening Pandora's Box . . . 133
7.2 Critical Analysis . . . 135
7.2.1 Cache and Directory Organization . . . 135
7.2.2 Conversion to RCtso . . . 135
7.2.3 Transparency of Genetic Programming . . . 136
7.3 Future Directions . . . 136
7.3.1 Microarchitectural Gaps and Power Modelling . . . 136
7.3.2 Better Formal Verification . . . 136

VI Appendix **139**

A Detailed Protocol Specification for TSO-CC **141**

A.1 Assumptions and Definitions 141

A.2 Protocol State Transition Tables 141

A.2.1 Private Cache Controller 142

A.2.2 Directory Controller 142

A.3 Additional Rules and Optimizations 145

A.3.1 Cache Inclusivity and Evictions 145

A.3.2 Timestamp Table Size Relaxations 145

A.3.3 Effect of L1 Timestamp update 145

A.3.4 Effect of L2 Timestamp update 146

A.3.5 TimestampReset Races 147

List of Tables

2.1 Definition of relations used to specify memory consistency models in the framework of [AMT14]. 16

4.1 TSO-CC specific storage requirements 49
4.2 System parameters for TSO-CC evaluation 58
4.3 Benchmarks and their input parameters for TSO-CC evaluation 59

5.1 RCtso ordering requirements 78
5.2 RC3 specific storage requirements 87
5.3 System parameters for RC3 evaluation 88
5.4 Benchmarks and their input parameters for RC3 evaluation 89
5.5 RC3 storage scaling 91

6.1 McVerSi Guest-Host interface 115
6.2 System parameters for McVerSi evaluation 118
6.3 McVerSi test generation parameters 119
6.4 McVerSi bug coverage 125
6.5 McVerSi bugs found up to 10 days 126
6.6 McVerSi maximum total transition coverage 126

A.1 TSO-CC private (L1) cache controller transition table 143
A.2 Directory (L2) controller transition table 144

List of Figures

1.1 Baseline architecture block diagram 4

2.1 Overview of specification and verification of multi-threaded programs using axiomatic memory consistency models. 15
2.2 Store buffering pattern showing forbidden executions 19
2.3 Simple mutual exclusion algorithm for SC 20
2.4 Message passing pattern showing forbidden executions 20
2.5 Relative optimization potential of various memory consistency models 23

3.1 Directory based MSI protocol transition diagram of stable states . . . 29
3.2 An example demonstrating the MSI protocol 30
3.3 Directory based MESI protocol transition diagram of stable states . . 32
3.4 Producer-consumer example with lazy RC 33

4.1 Producer-consumer and TSO ordering example 38
4.2 Message passing pattern. 48
4.3 TSO-CC storage scaling 60
4.4 TSO-CC execution times 63
4.5 TSO-CC network traffic 63
4.6 TSO-CC L1 cache misses 64
4.7 TSO-CC L1 cache hits and misses 64
4.8 TSO-CC self-invalidations upon data response 65
4.9 TSO-CC RMW latencies 66
4.10 TSO-CC breakdown of L1 self-invalidation cause 66

5.1 Extended producer-consumer example 76
5.2 RC3 execution times 92
5.3 RC3 network traffic 92
5.4 RC3 L1 cache misses 93

5.5 RC3 L1 cache hits and misses 93
5.6 RC3 normalized self-invalidations 94

6.1 Message passing pattern 108
6.2 McVerSi crossover and mutation example 113

Part I

Preamble

Chapter 1

Introduction

1.1 Cache Coherence Protocol Scaling

With the breakdown of Dennard scaling [Den+74]—the end of supply voltage scaling—architects have been looking for alternatives to take advantage of the continued increase in available transistor counts as per Moore's Law [Moo65]. At the beginning of the 21st century, the trend of the industry shifted towards using multiple cores on a single chip—"multicores" or chip multiprocessors (CMPs)—to improve performance for parallel workloads. Unfortunately, scaling CMPs to ever increasing core counts also faces several challenges [Esm+11]. A major challenge is the cache coherence protocol, as conventional approaches quickly outgrow the permitted cost to achieve ever increasing core counts [Cho+11; MHS12; KK10; Kel+10; RK12].

In shared memory multiprocessors, each processor typically accesses a local cache to reduce memory latency and bandwidth—Figure 1.1 illustrates a simple CMP baseline architecture. Data cached in local caches, however, can become out-of-date when they are modified by other processors. *Cache coherence* helps ensure shared memory correctness by making caches transparent to programmers, giving the illusion of a single shared address space. Shared-memory correctness is defined by the *memory consistency model* (MCM), which formally specifies in what order memory operations (reads and writes) must appear to the programmer [AG96; SHW11]: stricter models restrict reordering, whereas weaker models permit larger amounts of instruction reordering.

The relation between the processor's memory consistency model and the coherence protocol has traditionally been abstracted to the point where each subsystem considers the other as a black box [SHW11]. Generally, this is beneficial, as it reduces overall complexity; however, as a result, coherence protocols are designed to be compatible with

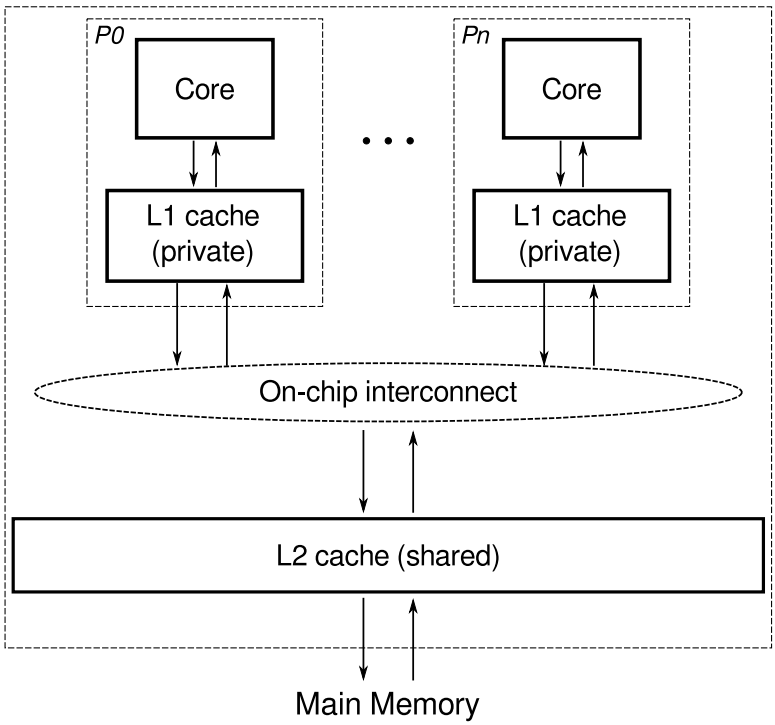


Figure 1.1: Block diagram of baseline architecture of chip multiprocessor (CMP) system assumed.

the strictest consistency model, sequential consistency (SC). SC mandates that writes are made globally visible before a subsequent memory operation. The coherence protocol plays a crucial role in the implementation: before writing to a cache line, conventional coherence protocols propagate writes *eagerly* by invalidating shared copies in other processors.

Providing eager coherence, however, comes at a cost. A simple approach is to use snooping-based protocols, where writes to non-exclusive cache lines are effectively broadcast. Unfortunately, in snooping-based systems, the interconnect quickly becomes a bottleneck with increasing number of processors [Aga+88]. A more scalable approach is to use directory-based protocols [CF78] in which the directory maintains, for each cache line, the list of processors caching that line, in the *sharing vector*. Upon a write to a non-exclusive cache line, invalidation requests are sent to only those processors caching that line.

While avoiding the potentially costly broadcasts, the additional invalidation and acknowledgement messages nevertheless represent overhead. More importantly, the bookkeeping to maintain the list of sharers becomes prohibitively expensive with increasing number of cores (in terms of area and power). Coherence protocols need to

overcome inherent scalability issues if we wish to continue the trend of ever increasing cores. More specifically, the size of the sharing vector increases linearly with the number of processors. With an increasing number of processors, it becomes prohibitively expensive to support a sharing vector for large-scale CMPs [Cho+11; MHS12; KK10; Kel+10; RK12].

In the foreseeable future, on-chip cache coherence will continue to play an important role in continuing to provide programmers an intuitive view of shared memory, but several *scaling challenges* need to be overcome [MHS12]:

- on-chip storage requirements;
- on-chip network traffic;
- inefficiencies caused by maintaining cache inclusion (e.g. false sharing);
- latency of cache misses;
- energy overheads.

Over the years, many approaches have addressed some of these challenges, especially by optimizing the data structures and cache organization [Cue+11; Fer+11; GWM90; MHS12; Pug+10; SK12; Wal92; ZSD10]. But the alternative, of optimizing the protocol itself, in particular for a particular memory consistency model, has not been explored sufficiently for modern systems. Recent years have seen renewed interest in this approach via *lazy* cache coherence protocols [ADC11; Cho+11; FC08; KK10; RK12; SKA13; SA15], highlighting its potential—however, these approaches target relaxed consistency models with explicitly exposed synchronization, e.g. Release Consistency (RC) [Gha95]. Lazy coherence protocols shift the task of invalidation from the writer to the reader via *self-invalidation* at synchronization boundaries [LW95].

This thesis focuses on the design of memory consistency directed cache coherence protocols, in particular for consistency models and architectures which *do not explicitly expose synchronization* to overcome some of the above challenges. This approach, however, makes it less straightforward to reason in terms of established properties of coherence (e.g. Single-Writer–Multiple-Reader [SHW11]): in the approach taken in this thesis, the property to be satisfied *is* the consistency model of the *full* system. In the final part of this thesis, a framework for fast simulation-based memory consistency verification is proposed to help designers verify *full-system* implementations of a design.

1.2 Memory Consistency and Cache Coherence

The memory consistency model specifies the permitted reordering of memory operations, with which the programmer can then reason about parallel program correctness [AG96; SHW11]. Although more relaxed memory consistency models are less intuitive from a programmer perspective, they can be motivated by performance optimizations in the implementation of a shared memory multiprocessor system. For example, write-buffers, in the absence of any other visible optimizations, give rise to Total Store Order (TSO), as in e.g. x86 [OSS09], where writes may appear to be reordered after following reads.

Broadly, the relationship between memory consistency model to cache coherence protocol is a specification to a part of its implementation; indeed, there are many other components in a system, e.g. core pipeline, and the combination of all components gives rise to the final consistency model. Here, the coherence protocol is responsible for the propagation of data in the memory hierarchy, in particular, between caches to make replicated data in independent caches appear as part of a single shared memory.

Yet, the classical definition of cache coherence is not tied too closely to the memory consistency model, but rather has various other definitions. One common definition is, that coherence must enforce the following invariant: per memory location, there must only be either a single writer (and no readers) or there may be several readers (but no writers)—also referred to as the Single-Writer–Multiple-Reader (SWMR) invariant [SHW11]. Indeed, this definition is sufficient for most traditional coherence protocols, as well as strict enough to guarantee compatibility with SC (in the presence of an in-order pipeline) [MS09].

As a result, conventional coherence protocols propagate writes *eagerly* by invalidating shared copies of data in other processors (before the write). To achieve this, protocols can either be snooping (interconnect bottleneck [Aga+88]) or use a directory tracking sharers (storage bottleneck [Cho+11; MHS12; KK10; Kel+10; RK12]). However, *modern CMPs no longer guarantee SC* (see ARM [AMT14; Flu+16], POWER [Sar+11], SPARC [SPA92], x86 [OSS09])—the main contributing factor being core pipeline optimizations, but not the coherence protocol.

Hypothesis: If modern CMPs no longer guarantee SC, is using coherence protocols designed to satisfy a model as strict as SC still required? And in relaxing this requirement, can more scalable protocols be designed? The main hypothesis of this thesis is, that *cache coherence protocols can be optimized for a target consistency model (weaker than SC), and in doing so address several of the scaling challenges—the main focus of this*

thesis being storage overheads while retaining baseline performance characteristics.

Finally, *verification* of cache coherence is an important aspect of design [ASL03]. Designing coherence protocols satisfying invariants such as SWMR (and thus be compatible with SC), is also beneficial for verification: indeed, using SWMR as a key invariant is straightforward to use in formal methods approaches such as model checking [PD97]. One direction is to design protocols with verifiability (using existing verification methodologies) in mind [Zha+14]. The other direction is to explore alternative verification methodologies to suit coherence protocols that may no longer satisfy SWMR. In particular, using the final system memory consistency model to be verified against, instead of indirect invariants. Furthermore, while verifying individual components like the coherence protocol is essential, the interaction between components (pipeline, coherence protocol, etc.) in a *full-system* should not be neglected. Currently, there is a lack of methodologies for memory consistency verification of a full-system in a pre-silicon environment (e.g. simulation): can we design a methodology that decreases the time to find as many bugs as possible pre-silicon (compared to alternatives)?

1.3 Contributions

Consistency Directed Cache Coherence Protocols: Existing lazy coherence cache coherence protocols [ADC11; Cho+11; FC08; KK10; RK12; SKA13; SA15] have limitations regarding portability. As they target relaxed consistency models explicitly exposing synchronization to the hardware, e.g. Release Consistency (RC) [Gha95], existing architectures with stricter models cannot benefit from them, as *legacy codes must continue to work*. Here, we consider the x86 and SPARC architectures which support variants of Total Store Order (TSO) [OSS09; SPA92]. The key challenge in designing a lazy coherence protocol for TSO is the absence of explicit *release* or *acquire* instructions; regular loads and stores have acquire and release semantics respectively.

The first contribution is **TSO-CC**, a coherence protocol that enforces TSO lazily without a full sharing vector. Without tracking sharers, the protocol must self-invalidate potentially stale cache lines upon potential acquires. In the most basic version of the protocol, every cache miss is assumed to be a potential acquire; in the *optimized* version of the protocol, we use *transitive reduction* to limit potential acquires, and therefore reduce costly self-invalidations. The use of a full sharing vector is an important factor that could limit scalability, which we overcome in our proposed protocol while *maintaining good overall performance in terms of execution times and on-chip network-traffic*.

TSO-CC's *storage overhead per cache line scales logarithmically with increasing core count*. More specifically, for 32 (128) cores, our best performing configuration reduces the storage overhead over a MESI baseline protocol by 38% (82%). Our experiments with programs from SPLASH-2, PARSEC and STAMP benchmarks show an average reduction in execution time of 3% over the baseline, with the best case outperforming the baseline by 19%.

Next, we explore a method to distinguish synchronization and data operations in the x86-64 architecture to exploit the explicit synchronization information that is already present in many recent language level memory consistency models (e.g. C11 [ISO11a], C++11 [ISO11b; BA08] and Java [MPA05]). The key contribution is **RC3**, a lazy cache coherence protocol for **RCtso**, and a seamless approach to adopt the protocol in the x86-64 architecture—thereby allowing the architecture to exploit the explicit synchronization information present in many recent language level memory consistency models. We achieve this by showing how to convey explicit ordinary and synchronization information to the hardware via a backward and forward compatible[1] ISA extension, and in doing so propose to change the consistency model from x86-TSO to x86-RCtso. The RC3 protocol then targets the RCtso consistency model lazily, without the need for a sharing vector nor per cache line timestamps. In comparison to a conventional lazy RC coherence protocol, RC3 achieves a 12% performance improvement on average owing to transitive reduction of redundant acquires using timestamps. In comparison to TSO-CC, RC3 reduces coherence storage requirements by 45% by eliminating per cache line L1 and L2 timestamps. Furthermore, eliminating per cache line timestamps also simplifies cache accesses as timestamps do not need to be tagged.

Full-System Memory Consistency Verification: Simulation of a design is available much earlier in the development cycle (pre-silicon). As such it is much cheaper if as many bugs as possible are found early. Furthermore, the added observability in simulation makes debugging more straightforward. For example, the advantages of simulation for memory system verification using user-guided random tests have been described and exploited by Wood et al. [WGK90]. Unfortunately, throughput (in terms of instructions executed in wall-clock time) of an accurate simulated system is orders of magnitude lower than a real chip. The challenge is, *how do we automatically generate efficient memory consistency tests for simulation, such that the wall-clock time to explore rare corner cases and find bugs is reduced?*

[1] Forward compatible meaning that new program codes can also still be run on old architectures.

Here, the focus lies on automated *simulation-based verification of a full-system design implementation*: we propose **McVerSi**, a test generation framework for *fast, coverage directed memory consistency verification in simulation*. Using a Genetic Programming (GP) [Koz92] based approach, we show how to generate tests for a full-system simulation that achieve improved test quality specifically for memory consistency verification, and also achieve greater coverage of the system (exploring rare corner cases): focusing on both these aspects leads to significantly reduced wall-clock time to find all studied bugs consistently (in comparison with alternative approaches). We evaluate McVerSi using the Gem5 [Bin+11] cycle accurate simulator in x86-64 full-system mode with Ruby (with implementations of a MESI variant and TSO-CC). Our evaluation also highlights the importance of verifying conventional protocols in the context of a full-system, as two new bugs in the MESI coherence protocol implementation of Gem5 have been found in the process.[2] In total, we study 11 bugs. In comparison with a pseudo-random test generator, and diy [Alg+12] generated litmus tests for TSO, our GP-based approach finds all bugs consistently within practical time bounds, thereby providing much higher bug finding guarantees.

1.4 Thesis Structure

The background required is summarized in the following Part II: Chapter 2 introduces memory consistency models, and Chapter 3 defines cache coherence protocols and their role in enforcing the consistency model. Part III introduces consistency directed cache coherence protocols, the main theme of this thesis: Chapter 4 proposes TSO-CC, followed by Chapter 5 introducing RC3. Both chapters are relatively self-contained, but for a greater appreciation of RC3, it is recommended that Chapter 4 (TSO-CC) be read first. The next part is on verification, and Chapter 6 proposes the McVerSi framework. Chapter 6 is sufficiently self-contained that Part III is not a necessary dependency, but is suggested to better understand the challenges McVerSi wishes to address, as well as details of the evaluation where TSO-CC is used as a case study. Finally, Chapter 7 concludes and provides perspectives on future directions.

[2] Fixes for the bugs have been sent to the Gem5 maintainers.

Part II

Background

Chapter 2

Memory Consistency Models

2.1 Overview

Programming shared memory multiprocessor systems correctly requires a precise definition of the *semantics* of such a system. In particular, the programmer must be aware of the memory access ordering guarantees the hardware provides. The memory consistency model (MCM) *formally specifies* the ordering guarantees with which the programmer can reason about parallel programs [AG96].

Over the years, various formalizations of MCMs have emerged. Both *axiomatic*—e.g. Sequential Consistency (SC) [Lam79], Total Store Order (TSO) [AMT14; SPA92], Release Consistency (RC) [Gha95; Gha+90], POWER [Alg+12; AMT14; Mad+12] and ARMv7 [AMT14]—as well as *operational* models—e.g. x86-TSO [OSS09], POWER [Sar+11]) and ARMv8 [Flu+16]—can be used to describe MCMs formally.

Axiomatic models present the model in terms of relations ordering operations as they may be observed, effectively forming a data-flow graph; these graphs should not be cyclic, as this would imply a contradiction and the programmer could no longer reason about their multi-threaded program. Operational models, on the other hand, are presented as abstract machines that if given a multi-threaded program, generate all allowable results. Both are important in their own right, as the former is more concise whereas the latter can give rise to more elegant proofs (e.g. Chapter 4 will introduce an operational model as a proof tool).

While there are various options for deciding upon a specific memory consistency model in a multiprocessor system, it is essential to find the right balance between programmability and performance. This section also provides an overview of the hierarchical approach to memory consistency, and how the choice of the memory

consistency model impacts the implementation [Gha+90; Adv93; AH90; Gha95].

2.2 Axiomatic Framework

This section summarizes the framework proposed by Alglave, Maranget, and Tautschnig [AMT14] to specify the axiomatic semantics of memory consistency models. We have chosen to use this framework, as it (1) is succinct while eliminating ambiguity (especially for the purpose of implementing decision procedures) and (2) is flexible enough to describe a variety of consistency models while relying on a library of common reusable definitions (hence a framework).

Frameworks have their disadvantages (no size fits all), but the advantages of relying on a proven framework with the ability to turn these definitions into a consistency model checker (required for Chapter 6) outweigh. Figure 2.1 illustrates the flow of how axiomatic consistency models are used in specification and verification of multi-threaded program behavior.

Notation: The models are described using classic set theory. As the relations capture order among events, we often use the more intuitive shorthand: $x \xrightarrow{\mathsf{rel}} y \triangleq (x,y) \in \mathsf{rel}$. The shorthand $\mathsf{r}_1;\mathsf{r}_2$ denotes the sequential composition of two relations: $(x,y) \in (\mathsf{r}_1;\mathsf{r}_2) \triangleq \exists z.(x,z) \in \mathsf{r}_1 \wedge (z,y) \in \mathsf{r}_2$. The predicate $\texttt{irreflexive}(\mathsf{r})$ is true iff r is irreflexive: $\texttt{irreflexive}(\mathsf{r}) \triangleq \neg(\exists x.(x,x) \in \mathsf{r})$. The predicate $\texttt{acyclic}(\mathsf{r})$ is true iff the transitive closure of r is irreflexive: $\texttt{acyclic}(\mathsf{r}) \triangleq \texttt{irreflexive}(\mathsf{r}^+)$. The reflexive-transitive closure of r is written as r^*.

2.2.1 Instruction Semantics

An ISA may have a diverse set of instructions, and different instructions when executed may result in the same *effect on the memory system*. Given a multi-threaded program, and some instruction semantics, each instruction is mapped to a set of *events*. This helps to avoid redundancies and simplify the presentation of a particular memory consistency models.

Definition 2.1 (Events)**.** Events are unique, referred to with a lower case letter (e.g. e). An event captures the thread ($\mathsf{proc}(e)$), address ($\mathsf{addr}(e)$) and action. For all events in a program, WR, WW, RR, RW are the relations capturing all write-read, write-write, read-read and read-write pairs respectively.

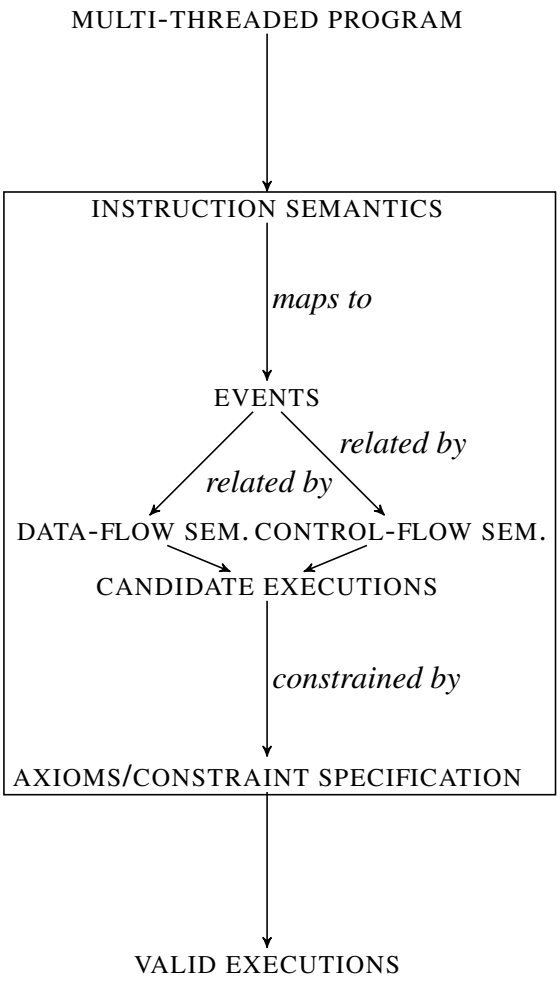


Figure 2.1: Overview of specification and verification of multi-threaded programs using axiomatic memory consistency models.

To ease the rest of the discussion, unless otherwise specified, we will assume the following generic mapping:

- Loads map to exactly one read event.
- Stores map to exactly one write event.

As much of the discussion is not specific to a particular ISA, we will refer to reads and writes rather than loads and stores where the mapping is obvious from the context. For a complete account of instruction semantics we refer to reader to [AMT14].

2.2.2 Candidate Executions

A candidate execution captures one possible execution of a multi-threaded program. Depending on the control-flow and data-flow semantics, there may be a large set of

Table 2.1: Definition of relations used to specify memory consistency models in the framework of [AMT14].

Relation	*Name*	*Source*	*Subset of*	*Definition*
po	program-order	execution	$\mathbb{E} \times \mathbb{E}$	instruction order lifted to events
rf	read-from	execution	WR	links a write w to a read r taking its value from w
co	coherence	execution	WW	total order over writes to the same memory location
ppo	preserved program order	architecture	po	program order maintained by the architecture
fences	fences	architecture	po	events ordered by fences
prop	propagation	architecture	WW	order in which writes propagate
po-loc	program order subset of same address events	derived	po	$\mathsf{po\text{-}loc} \triangleq \left\{(x,y) \mid x \xrightarrow{\mathsf{po}} y \wedge \mathsf{addr}(x) = \mathsf{addr}(y)\right\}$
com	communications or conflict orders	derived	$\mathbb{E} \times \mathbb{E}$	$\mathsf{com} \triangleq \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$
rfe	read-from external	derived	rf	$\mathsf{rfe} \triangleq \left\{(w,r) \mid w \xrightarrow{\mathsf{rf}} r \wedge \mathsf{proc}(w) \neq \mathsf{proc}(r)\right\}$
fr	from-read	derived	RW	links reads to writes based on observed rf and co: $\mathsf{fr} \triangleq (\mathsf{rf}^{-1}; \mathsf{co}) \triangleq \left\{(r,w) \mid \exists w'.w' \xrightarrow{\mathsf{rf}} r \wedge w' \xrightarrow{\mathsf{co}} w\right\}$
fre	from-read external	derived	fr	$\mathsf{fre} \triangleq \left\{(r,w) \mid r \xrightarrow{\mathsf{fr}} w \wedge \mathsf{proc}(r) \neq \mathsf{proc}(w)\right\}$
hb	happens before	derived	$\mathbb{E} \times \mathbb{E}$	$\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$

possible executions (but not all of which may be valid as per the constraint specification).

Definition 2.2 (Candidate executions)**.** A candidate execution is a tuple $(\mathbb{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co})$, with the set of events $\mathbb{E}$ in the program, and the relations program-order po (via control-flow semantics), reads-from rf and coherence (or write-serialization) order co (via data-flow semantics); see Table 2.1 for details.

2.2.3 Architecture Definitions

An *architecture* defines the architecture specific details of a particular memory consistency model, and is used by the constraint specification to decide if a particular execution is valid or not. An architecture is defined by the tuple of relations $(\mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$, preserved program-order ppo, fences and propagation prop; see Table 2.1 for details. In the following we will provide the instantiations of Total Store Order (TSO) and Sequential Consistency (SC) as provided by [AMT14]. Both models are discussed informally in §2.4.

Definition 2.3 (Sequential Consistency)**.** No relaxations are permitted and all four possible instruction orderings (read-read, read-write, write-read, write-write) must be maintained.

$$\begin{aligned} \mathsf{ppo} &\triangleq \mathsf{po} \\ \mathsf{fences} &\triangleq \emptyset \\ \mathsf{prop} &\triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rf} \cup \mathsf{fr} \end{aligned}$$

Definition 2.4 (Total Store Order)**.** Only the write to read ordering is relaxed. Reads to the same address as a preceding writes w by the same thread must observe either w or a write by another thread that happened after w in co; this, however, has no effect on the required visibility by other threads, which implies that prop only includes rfe (and not rf). The relation mfence captures write-read pairs separated by a fence instruction.

$$\begin{aligned} \mathsf{ppo} &\triangleq \mathsf{po} \setminus \mathsf{WR} \\ \mathsf{fences} &\triangleq \mathsf{mfence} \\ \mathsf{prop} &\triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{fr} \end{aligned}$$

2.2.4 Constraint Specifications

Given a candidate execution and an architecture specification, the constraints (or axioms) decide if the execution is valid under the complete model.

Definition 2.5 (SC PER LOCATION)**.** Satisfied if $\texttt{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{com})$. SC PER LOCATION ensures that communications/conflict orders com cannot contradict program-order per memory location po-loc.[1]

Definition 2.6 (NO THIN AIR)**.** Satisfied if $\texttt{acyclic}(\mathsf{hb})$. This constraint ensures that the happens-before order hb, which captures preserved program-order ppo, fenced

[1] This constraint by itself is one of the definitions of coherence, as discussed in §3.2.

instructions fences, but also reads from other threads rfe is not contradictory. Effectively, this prevents reads from observing values "out of thin air", i.e. before they appear to have been written by some other thread.

Definition 2.7 (OBSERVATION)**.** Satisfied if $\texttt{irreflexive}(\mathsf{fre};\mathsf{prop};\mathsf{hb}^*)$. OBSERVATION constrains the values a read may observe. Assume a write w_a to address a, a write w_b to address b, which are ordered in prop ($w_a \overset{\mathsf{prop}}{\rightarrow} w_b$), and a read r_b reading from w_b ($w_b \overset{\mathsf{rf}}{\rightarrow} r_b$), then any read r_a that happens after r_b ($r_b \overset{\mathsf{hb}}{\rightarrow} r_a$) cannot read from a write before w_a.

Definition 2.8 (PROPAGATION)**.** Satisfied if $\texttt{acyclic}(\mathsf{co} \cup \mathsf{prop})$. This constraint imposes restrictions on the order in which writes are propagated to other threads, i.e. ensuring that the propagation order prop does not contradict coherence (write-serialization) order co. This constraint and depending on the prop order defines if the consistency model is *write/multi-copy atomic*, i.e. if all threads observe writes in the same order or not.

2.3 Assumptions on Progress Guarantees

It should be noted that most models focus on ordering constraints, however, *when* memory operations propagate to other threads is usually not explicit: unless specified, it is implicitly assumed that operations *eventually* become visible (*eventual write propagation*); furthermore, it is sufficient for only conflicting operations, i.e. synchronization, to eventually become visible [Gha95]. This guarantee is necessary for programs to make useful progress.

Indeed, when reasoning about a concurrent system, two types of properties should be proved: *safety* ("nothing bad will happen") and *liveness* ("something good will happen eventually") [Lam77; AS85]. In the context of multithreaded programs, the programmer asserts safety with the help of the ordering guarantees imposed by the memory consistency model, but liveness also requires guarantees on eventual write propagation.

Few consistency models place a concrete time bound or fairness restrictions on write propagation, implying that programmers must design algorithms taking this into account. A notable exception is the bounded staleness TSO model (TSO$[S]$) proposed by Morrison and Afek [MA14].

2.4 System-Centric Models

System-centric memory consistency models expose the direct interface with the hardware. In more relaxed consistency models, it becomes more difficult for programmers to reason about parallel programs, and as such, stricter models are preferred when programmers are expected to reason at the hardware level. This section briefly summarizes three system-centric models which will be central to later chapters.

2.4.1 Sequential Consistency

Sequential Consistency (SC) was first defined by Lamport [Lam79]. In SC, no relaxations are permitted and all four possible instruction orderings (read-read, read-write, write-read, write-write) must be preserved. In the words of Lamport [Lam79], a multiprocessor system can be called *sequentially consistent* if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." See Definition 2.3 for the formal definition.

Consider the pattern in Figure 2.2. Here, SC forbids the case where both reads return the initial value. With a simple mutual exclusion algorithm (e.g. Dekker follows this pattern [Sco13]) as shown in Figure 2.3, this implies that the critical section cannot be entered by both threads at the same time.

init: x = 0, y = 0	
Thread 1	**Thread 2**
$x \leftarrow 1$	$y \leftarrow 1$
$r1 \leftarrow y$	$r2 \leftarrow x$
SC forbids: $r1 = 0 \wedge r2 = 0$	

Figure 2.2: Store buffering pattern showing forbidden executions.

2.4.2 Total Store Order

Since Total Store Order (TSO) is the memory consistency model found in x86 processors, its various implementations have been analyzed in detail [OSS09]. Historically, TSO has been the result of taking writes out of the critical path and entering committed writes into a FIFO write-buffer, potentially delaying a write to cache or other parts of

init: x = 0, y = 0	
Thread 1	**Thread 2**
x ← 1	y ← 1
if (y == 0) then	if (x == 0) then
critical section	critical section
else ...	else ...

Figure 2.3: Simple mutual exclusion algorithm for SC [Lam79].

init: x = 0, y = 0	
Thread 1	**Thread 2**
x ← 1	r1 ← y
y ← 1	r2 ← x
SC & TSO forbids: $\mathsf{r1} = 1 \wedge \mathsf{r2} = 0$	

Figure 2.4: Message passing pattern showing forbidden executions.

the memory hierarchy [OSS09; SC97]. Reads to the same address as prior writes by the same processor must not be affected, which mandates that reads *bypass* the write-buffer. Consequently, this relaxes the write to read ordering. See Definition 2.4 for the formal definition. As every write can potentially be a release, each write needs to *eventually* propagate to other processors, so that they are made visible to a matching acquire and forward progress is possible (see §2.3).

Figure 2.4 illustrates a common pattern where TSO (and SC) still provides intuitive semantics due to maintaining the write-write ordering. However, since write-read is now relaxed, the case forbidden under SC in Figure 2.2 is indeed valid under TSO. Consequently, this implies that the simple mutual exclusion algorithm in Figure 2.3 no longer works correctly; to make such an algorithm work again, modern architectures that provide variants of TSO, e.g. x86, need to extend TSO to provide explicit fence instructions enforcing ordering where required.

Furthermore, TSO guarantees that writes become visible to other threads at the same time—this is also referred to as *write atomicity* or *multi-copy atomicity*. This is what differentiates TSO from a related model, Processor Consistency (PC) [Gha+90], which does not mandate write/multi-copy atomicity.

2.4.3 Release Consistency

A relaxed and relatively simple model which explicitly exposes synchronization operations via special instructions is Release Consistency (RC) [Gha95; Gha+90]. In RC, special *release* and *acquire* instructions are used to enforce an ordering with other memory operations in program order.

Given a write-*release*, all memory operations prior must be visible before the write; a read-*acquire* enforces preserving the program ordering of all operations after it. In addition, releases guarantee eventual propagation of synchronization data so that they become visible to corresponding acquires. The particular RC-variant imposes additional restrictions on ordering between synchronization, e.g. RCsc requires that all possible orderings between synchronization are maintained.

Definition 2.9 (RCsc)**.** Let AM be the relation linking all acquires a to all events e: $\mathsf{MS} \triangleq \{(a,e) \mid (a,e) \in \mathbb{E} \times \mathbb{E} \wedge \texttt{is\_acquire}(a)\}$. Let MS be the relation linking all events e to a write-release s: $\mathsf{MS} \triangleq \{(e,s) \mid (e,s) \in \mathbb{E} \times \mathbb{E} \wedge \texttt{is\_release}(s)\}$. RCsc is instantiated as follows:

$$\begin{aligned} \mathsf{ppo} &\triangleq (\mathsf{po} \cap \mathsf{AM}) \cup (\mathsf{po} \cap \mathsf{MS}) \\ \mathsf{fences} &\triangleq \emptyset \\ \mathsf{prop} &\triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{fr} \end{aligned}$$

With RC, if the operations in the pattern of Figure 2.4 were all marked as ordinary, all executions would be valid. To only observe the same valid executions as TSO (or SC), it would be required to explicitly mark the write and read of y as release and acquire respectively.

2.5 Programmer-Centric Models

While many commercial multiprocessor systems adopt very relaxed memory consistency models, giving architects fewer restrictions on optimizations, this usually complicates reasoning about parallel programs at the hardware level. This problem, however, can be solved if we assume that the programmer does not need to reason about programs using the system-centric consistency models, and instead is exposed to a higher level abstraction at the programming language level [AG96].

The only requirement of the hardware level consistency model then is that any language level consistency model can be mapped to the hardware level. The formal basis

for this approach can be found in Adve et al.'s *data-race-free* [Adv93; AH90; AH93] (DRF) and Gharachorloo et al.'s *properly-labeled* [Gha95; Gha+90] (PL) models. In essence, the programmer explicitly labels synchronization and data operations correctly. In return, the system (compiler and hardware) guarantees SC—often referred to as *SC for DRF*.

Modern programming languages are converging towards clearly defined memory consistency models, and as such, the programmer only needs to reason in terms of the language level consistency model. For instance, C++11 is an adaptation of *data-race-free-0* [BA08]. However, for hardware to be able to benefit from the explicit synchronization information, the hardware's consistency model should be able to distinguish between synchronization and data operations. A straightforward implementation of *data-race-free-0* is using RC [Gha+90] (without nsync) [Adv93], where data operations are mapped to ordinary loads and stores, and synchronization operations are mapped to acquires and releases.

As a result, the hardware benefits from additional opportunity for optimization, and in particular, coherence protocol implementations can be *lazy* (§3.4). Propagation of ordinary memory operations can be delayed until an order can be re-established at synchronization boundaries [DSB86; KSB95; SD87]. This permits the protocol to remove the costly data structures to maintain a list of sharers, i.e. the sharing vector, and instead rely on self-invalidation upon synchronization boundaries as demonstrated by numerous prior works [KSB95; Cho+11; RK12; SKA13; KCZ92].

Finally, Figure 2.5 provides an overview of various memory consistency models and their relative optimization potential. Weaker consistency models generally present more opportunity for hardware optimizations. An efficient mapping from language to hardware level can be achieved, i.e. preserving as much opportunity for optimization as possible, if the system-centric model can preserve at least as much information about allowable memory operation reordering as the programmer-centric model.

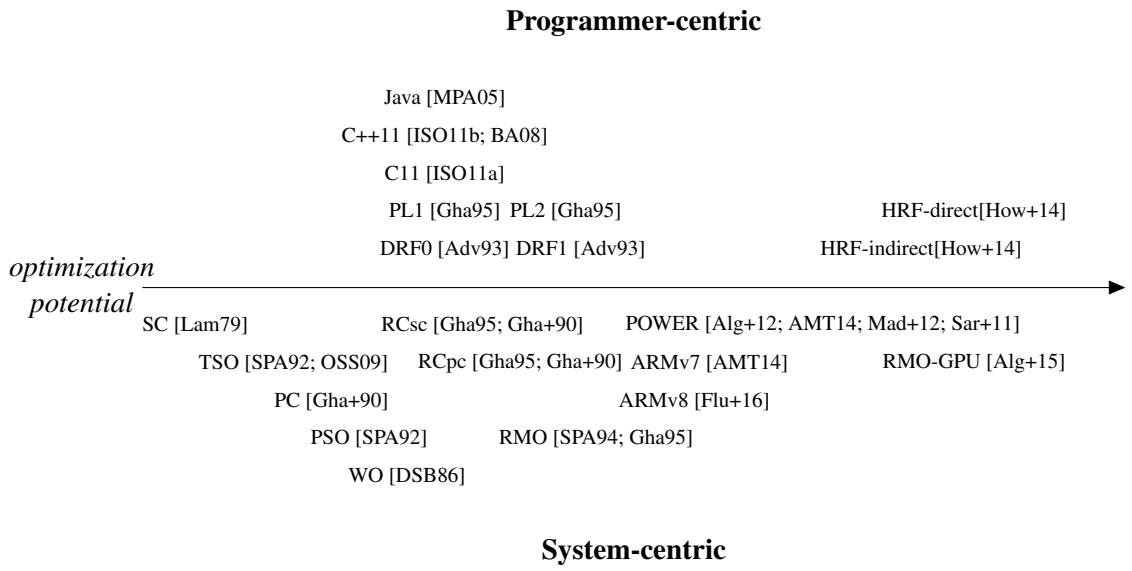


Figure 2.5: Relative optimization potential of various memory consistency models. An efficient mapping from language to hardware level can be achieved if the system-centric model can preserve at least as much information about allowable memory operation reordering as the programmer-centric model.

Chapter 3

Cache Coherence Protocols

3.1 Overview

Since modern multiprocessors use caches to hide memory access latencies (assuming an architecture like in Figure 1.1), each core may have its own local cache (L1). With separate L1s caching the same data, modifications to this data may result in a different view of these memory locations. The role of cache coherence is then, to ensure that caches become invisible (as is the case in a uniprocessor) to the programmer. Ultimately, the programmer should only worry about the memory consistency model to write correct parallel programs as discussed in Chapter 2. As a corollary, we may argue that the cache coherence protocol is a vital component in the enforcement of the target consistency model. However, there exist several definitions of cache coherence, which we discuss in §3.2. Following that, an overview of various classes of implementations for coherence protocols are discussed (§3.4).

3.2 Definition of Coherence

Classical Definitions: What exactly is *coherence*, and what is the relationship to the memory consistency model? A variety of (partly overlapping) definitions exist [SHW11; Gha95; HP07], on what may be called a "cache coherence protocol."

Definition 3.1 (Coherence Invariants: SWMR and DV [SHW11])**.** Sorin, Hill, and Wood [SHW11] establish the following invariants to be satisfied by a coherent system.

1. ***Single-Writer–Multiple-Reader (SWMR) Invariant.*** *For any memory location A, at any given (logical) time, there exists only a single core that may write to A*

(and can also read it) or some number of cores that may only read A.

2. ***Data-Value (DV) Invariant.*** *The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.*

The above definition is very implementation centric, but sufficient to capture the essential properties of traditional eager coherence protocols. Sticking to this definition, however, does not make it intuitive to design lazy coherence protocols (§3.4). Indeed, some of these protocols may violate the above definition but still satisfy the below definitions and/or the target memory consistency model (as we will see later).

Definition 3.2 (Coherence guarantees SC per memory location [SHW11])**.** Furthermore, Sorin, Hill, and Wood [SHW11] provides the following alternative consistency-like definition of coherence: *a coherent system must appear to execute all threads' loads and stores to a single memory location in a total order that respects the program order of each thread.*

This definition is arguably much more abstract than Definition 3.1, but also much broader in its scope for optimization. In fact, this definition is part of even the weakest consistency models [AMT14, §4.2 SC PER LOCATION]. For example, although SWMR is violated by the protocols in Part III, Definition 3.2 is not (as otherwise TSO would have been violated).

Definition 3.3 (Coherence propagates and serializes per-location writes [SHW11; Gha95])**.** As defined by [Gha+90; Gha95] and summarized in [SHW11], coherence guarantees:

(1) every store is eventually made visible to all cores;

(2) writes to the same memory location are serialized (i.e., observed in the same order by all cores).

This definition again seems broader than the previous definitions. Here, the inclusion of eventual write propagation is arguably an important insight, as it is necessary to guarantee useful progress (liveness) of multithreaded programs (see §2.3). The second clause on write serialization is only a necessary condition for achieving SC per location (Definition 3.2), but not equivalent; crucially, it lacks any notion of program order.

Definition 3.4 (Coherence preserves program order, propagates and serializes per-location writes [HP07])**.** Hennessy and Patterson [HP07] defines coherence as follows: *a memory system is coherent if*

(1) a read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P;

(2) a read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses;

(3) writes to the same location are serialized; [. . .].

Finally, the last definition given by Hennessy and Patterson [HP07] largely overlaps with Definition 3.3. Requirement (2) of a read returning a write if they are "sufficiently separated in time" can be likened to the eventual write propagation requirement (1) of Definition 3.3. Constraint (3) of Definition 3.4 is equivalent to (2) of Definition 3.3.

Any of these definitions may be seen as the most basic *specification* of the properties that any given protocol must meet to be a correct coherence protocol implementation. In particular, as Definition 3.1 is formulated as an invariant that does not depend on the full execution history of a system, but rather in terms of access permissions or limited state, this definition seems more intuitive for a designer of a protocol. For this reason, unlike the consistency-like Definitions 3.2, 3.3 and 3.4, the use of Definition 3.1's invariants also make formal verification (e.g. model checking) of coherence protocols a more tractable problem (see §6.7).

The above definitions seem sufficient to capture the properties of *conventional hardware coherence protocols*, without having to understand the promised memory consistency model of the system in detail. And in turn, these definitions are strict enough, that meeting them provides compatibility with all common consistency models, in particular the lowest common denominator, SC. Of course, the final consistency model is subject to more than just the coherence protocol, especially the core's pipeline (e.g. write-buffers in TSO; see §2.4.2). But, with a core pipeline presenting loads and stores in program order to the memory system, and a coherence protocol satisfying the above definitions, the final consistency model provided will be SC [MS09].

Blurring the Line: However, certain protocols may violate some of the above invariants without violating the desired consistency model. For example, lazy coherence protocols

(§3.4) can violate the SWMR invariant, but still satisfy the target consistency model (albeit usually not SC, but rather some form of SC for DRF; see §2.5). Despite such protocols breaking the established norm, it has been argued that this new class of protocols can still be called a cache coherence protocol [Cho+11; SKA13; SA15; SAA15], as their primary objective ("making caches transparent") is unchanged; indeed, they satisfy at least Definition 3.3 [SA15].

Other memory systems, such as those in GPUs, have been called *incoherent* as no conventional hardware cache coherence mechanism is present [Sin+13], or have been referred to as possessing software-based coherence mechanisms [SAA15]. Yet, such systems still have definable memory consistency models [How+14; Alg+15].

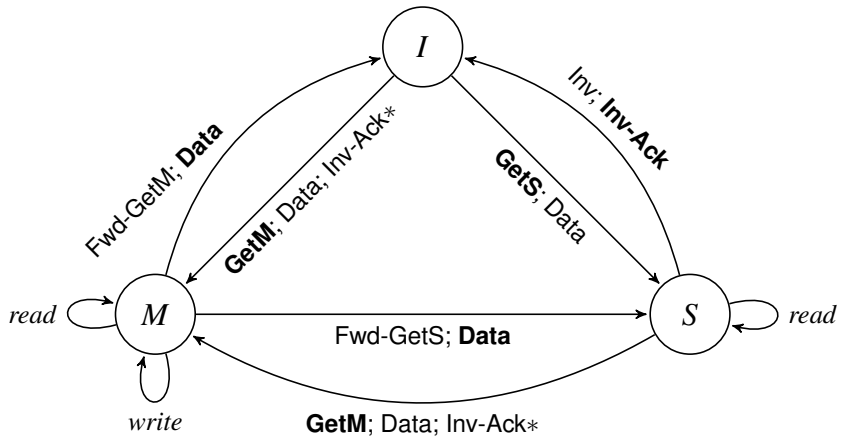
If the coherence protocol provides such a crucial component in the implementation of the consistency model, then why not use the target consistency model as the specification, i.e. *go beyond the per-location rules?* This being the main theme of this thesis, we will explore this idea in detail later. It shall be noted, however, that this approach may also bring with it a new set of challenges.

3.3 Baseline and Assumptions

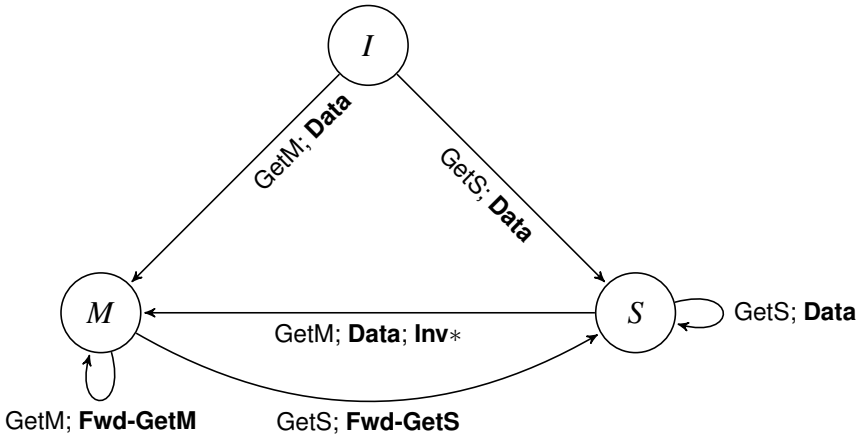
Throughout this thesis, the architecture of Figure 1.1 is assumed. Note that the interconnection network is an *unordered network* and as such the baseline considered is a system with a *directory-based coherence protocol*, which requires maintaining a list of sharers or sharing vector. The alternative, snooping-based protocols, are typically used in systems with smaller processor counts [Aga+88]; these protocols effectively rely on broadcasts and the interconnection network (e.g. bus) to have certain ordering properties. In the following we will introduce a variant of a MSI eager directory coherence protocol. For a detailed account of various standard coherence protocols, we refer the reader to the primer by Sorin, Hill, and Wood [SHW11].

Figure 3.1 shows the transition diagram of a variant of a directory-based MSI protocol. Evictions have been omitted to keep the diagram readable; we will assume silent evictions from the Shared state. Furthermore, we will also assume that all messages travel on unordered networks, which also necessitates extra transient states and acknowledgement messages (not shown).

Figure 3.2 illustrates how this variant of the MSI protocol ensures coherence with two processors and a shared L2 cache (with an embedded directory). In this case the producer-consumer program of Figure 3.2a assumes a strict consistency model, such as



(a) MSI cache controller transition diagram of stable states.



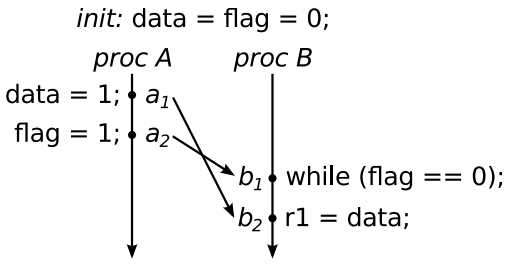
(b) MSI directory controller transition diagram of stable states.

Figure 3.1: Directory based MSI protocol transition diagram of stable states (evictions omitted). Each transition is labeled with a set of messages **sent (bold)** or received, where a $*$ denotes zero or more messages; the first message is the trigger of the transition or initiated by the controller due to a read (GetS) or write (GetM).

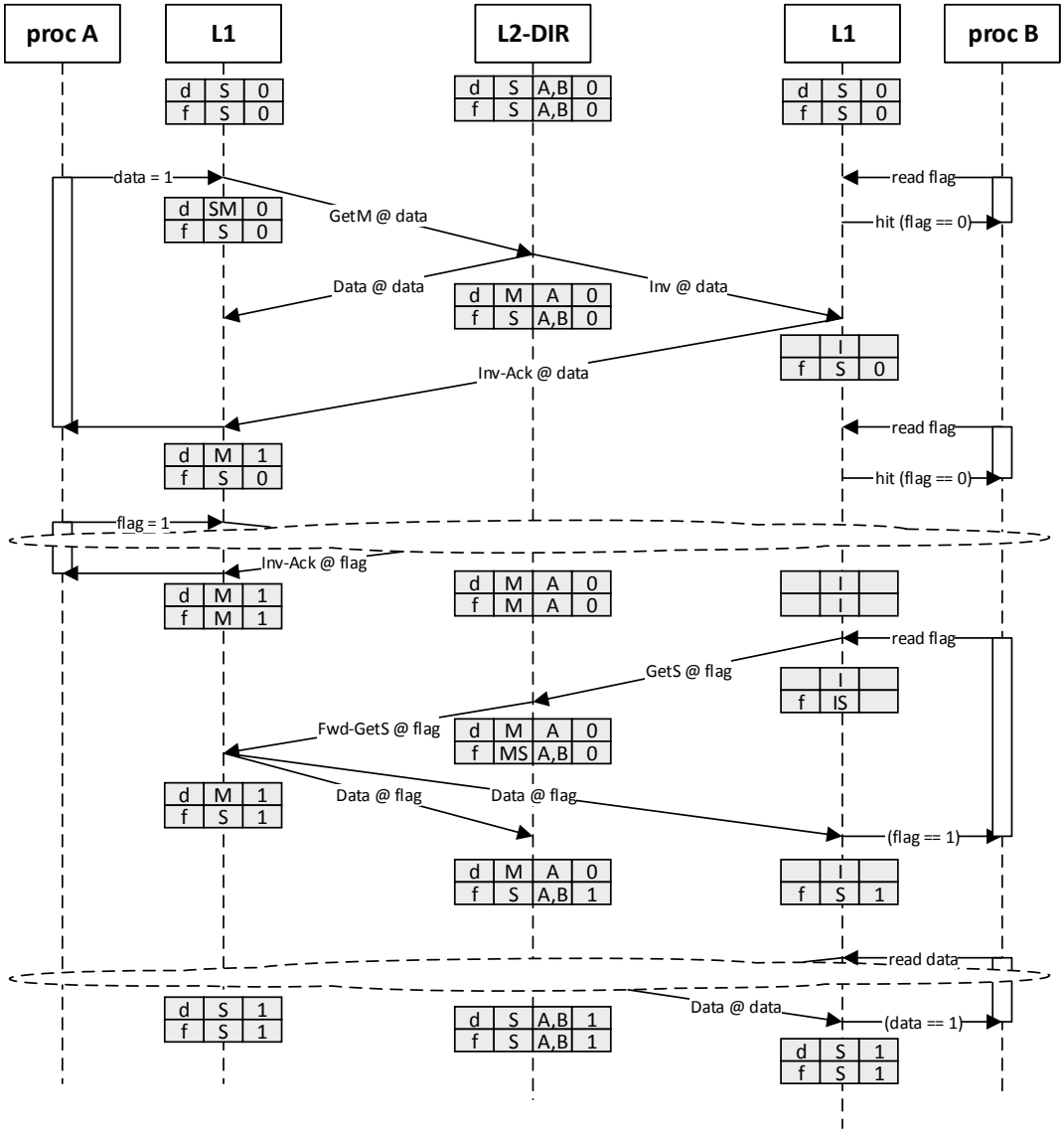
SC (or TSO), which this MSI protocol satisfies (assuming an in-order pipeline).

Initially, assume that the variables are both cached in Shared state in both L1s. Upon a_1, the write request from processor A triggers a GetM for data to be sent to the L2 directory. Upon receipt of the GetM message in the L2, invalidations are sent to all other sharers (proc. B), the line is transitioned to Modified and a response is sent to the requester. Receipt of the invalidation message by B is responded with by an acknowledgement to A, which can then complete the write. Similarly for the write of flag at a_2.

The read of flag at b_1 misses at this point, due to being previously invalidated. This causes a GetS message to be sent to the L2, which updates the sharers to include B and



(a) Producer-consumer example. A simple example, in which the thread in proc. B shall only access data after the write of flag completed in proc. A.



(b) Example trace of producer-consumer example with a MSI protocol. Each cache maintains: tag (in the figure abbreviated with letter of variable name), state, list of sharers (directory only), and the current data value.

Figure 3.2: An example demonstrating the MSI protocol.

then forwards the request to processor A. The request is responded to by processor A with a data message to both L2 and the initial requester. Upon receipt of the updated data, the L2 transitions the line back to Shared again, and could now satisfy further incoming GetS requests. Finally, processor B also receives the updated data, and observes the value of flag to be 1. Similarly for the read of data at b_2 which observes the correct value of 1.

Crucially, note that a write will stall until all other sharers have been invalidated, and the write is effectively visible to all other processors. Hence writes are propagated eagerly.

3.3.1 Adding the Exclusive State

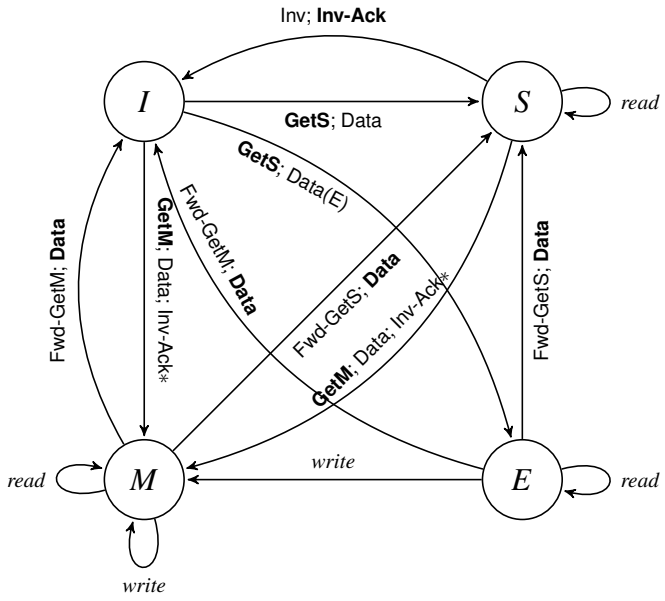
Unfortunately, with the shown MSI protocol, pathologies exist for e.g. private data, where a thread may first issue a read request (GetS) followed by a write request which is not silent (GetM) causing excessive network traffic and increased latency. An optimization to this problem is adding the Exclusive state, which permits the L2 to respond with exclusive data if the line is not cached in another L1. Thus, a write to a cache line obtained in the Exclusive state is silent, avoiding the additional latency. Figure 3.3 shows the transition diagram of a variant of a directory-based MESI protocol, which extends the above MSI protocol. We will refer to this protocol as the *MESI baseline protocol* henceforth.

3.4 Eager versus Lazy Coherence

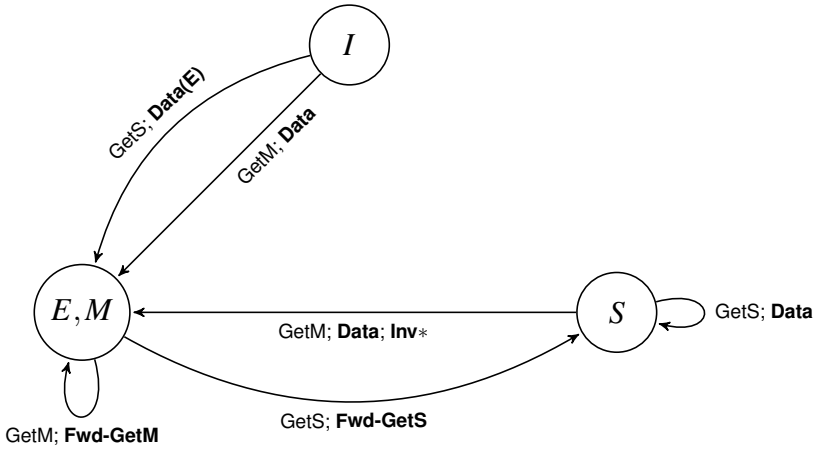
Let us establish the relationship between the coherence protocol and consistency model. Given a target memory consistency model, the coherence protocol must ensure that memory operations become visible according to the ordering rules prescribed by the consistency model.

In SC, because all orderings are enforced, and in particular the write-read ordering, a write must be made visible to all processors before a subsequent read. This requirement is ensured via the use of *eager coherence* protocols which propagate writes eagerly by invalidating or updating shared cache lines in other processors [SD87].

On the other hand, if the consistency model is relaxed, i.e. not all possible orderings between memory operations are enforced, propagation of unordered memory operations can be delayed until an order can be re-established through synchronization bound-



(a) MESI cache controller transition diagram of stable states.



(b) MESI directory controller transition diagram of stable states.

Figure 3.3: Directory based MESI protocol transition diagram of stable states (evictions omitted). Each transition is labeled with a set of messages **sent (bold)** or received, where a $*$ denotes zero or more messages; the first message is the trigger of the transition or initiated by the controller due to a read (GetS) or write (GetM).

aries [DSB86; KSB95; LW95; SD87]. In other words, *lazy coherence* protocols exploit the fact that relaxed consistency models require memory to be consistent only at synchronization boundaries. Using an eager coherence protocol in a system implementing a relaxed consistency model is potentially wasteful, as employing a lazy approach to coherence opens up further optimization opportunities to remedy the shortcomings of eager coherence protocols, as demonstrated by [Cho+11; KSB95; RK12; SKA13] in

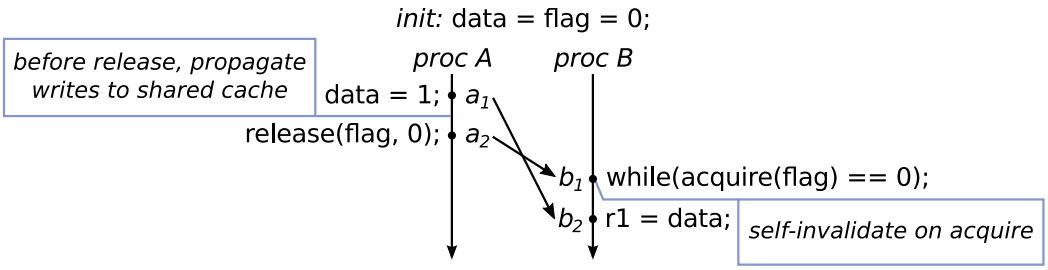


Figure 3.4: Producer-consumer example with lazy RC.

the context of consistency models like RC (explicit synchronization).

The approach was first proposed for distributed shared memory (DSM) coherence for Lazy Release Consistency [CBZ91; Kel+94; KCZ92]. In DSM systems, the high latency of message exchanges between nodes has been the key driver behind favoring consistency models that permits an implementation to delay and buffer write propagation as long as possible. As such, it is unsurprising that lazy implementations of relaxed consistency models, such as RC, have initially been explored in the context of DSM systems.

RC (and variants) provides the optimal constraints (explicit synchronization) for a lazy coherence approach and as such is the only consistency model for which lazy coherence approaches have been studied in great detail. Figure 3.4 illustrates how the producer-consumer would need to be modified for an RC system. Typically, in a system supporting RC, lazy coherence can be implemented by (1) propagating release writes and ensuring that all writes before the release are propagated first and (2) upon an acquire, self-invalidating all locally cached shared data.

Part III

Consistency Directed Cache Coherence Protocols

Chapter 4

TSO-CC: Consistency Directed Cache Coherence for TSO

4.1 Introduction

Although there have been a number of approaches to more scalable coherence purely based on optimizing eager coherence protocols and cache organization [Cue+11; Fer+11; GWM90; MHS12; Pug+10; SK12; Wal92; ZSD10], we are interested in an alternative approach. *Lazy coherence* has generated renewed interest [ADC11; Cho+11; RK12], as a means to address the scalability issues of coherence protocols. Lazy coherence protocols exploit the insight that relaxed consistency models such as RC require memory to be consistent only at synchronization boundaries (see §3.4). Consequently, instead of eagerly enforcing coherence at every write, coherence is enforced lazily only at synchronization boundaries. Thus, upon a write, data is merely written to a local write-buffer, the contents of which are flushed to the shared cache upon a *release*. Upon an *acquire*, shared lines in the local caches are self-invalidated—thereby ensuring that reads to shared lines fetch the up-to-date data from the shared cache. In effect, the protocol is much simpler and *does not require a sharing vector*. Indeed, designing hardware with a specific programming model—exposing more information about the software—in mind can lead to simpler hardware, demonstrated in the context of coherence protocols as early as [Hil+92].

It follows that, in contrast to conventional techniques for enhancing scalability, lazy coherence protocols have an added advantage since they are memory consistency directed.

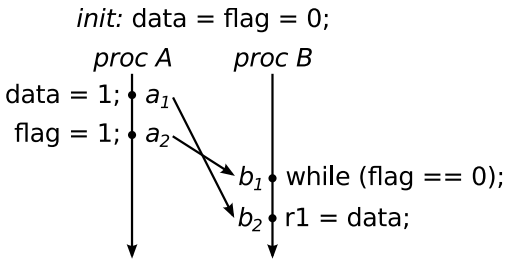


Figure 4.1: In TSO architectures that employ conventional (eager) coherence protocols (as illustrated with Figure 3.2), (1) eager coherence ensures that the write a_2 to flag from proc. A becomes visible to proc. B without any additional synchronization or memory barrier. (2) Once b_1 reads the value produced by a_2, TSO ordering ensures that the read b_2 sees the value written by a_1.

4.1.1 Motivation

However, one important limitation of existing lazy coherence protocols concerns portability. Since they only enforce relaxed consistency models such as RC, they are not directly compatible with widely prevalent architectures such as x86 and SPARC which support variants of TSO (see §2.4.2). Thus, legacy programs written for TSO would break on architectures that employ current lazy coherence protocols for RC.

Yet we conjecture that, due to TSO relaxing the write-read ordering, the coherence protocol is amenable to an efficient lazy coherence approach. This follows from the fact that writes in typical TSO implementations retire into a local write-buffer and are made visible to other processors in a lazy fashion already. Consequently, making them visible to other processors as soon as the writes exit the write-buffer is overkill.

4.1.2 Requirements

The key challenge in designing a lazy coherence protocol for TSO is the absence of explicit *release* or *acquire* instructions. Indeed, since all reads have acquire semantics and all writes have release semantics, simple writes and reads are used for synchronization. This is illustrated in the producer-consumer example in Figure 4.1, where the write a_2 is used as a release and the read b_1 is used as an acquire. In TSO, since any given write can potentially be a release, it is essential that each write is *eventually propagated* to other processors so that the value written is eventually made visible to a matching acquire. In the above example, the value written by a_2 should be made visible to b_1—or else, processor B will keep spinning indefinitely and will never make forward progress.

Recall that TSO mandates enforcing all possible orderings except the write-read order. In the above example, once b_1 reads the value produced by a_2, TSO ordering implies that b_2 reads the value written by a_1. One way to trivially ensure TSO is to consider each read (write) to be an acquire (release) and naïvely use the rules of a lazy RC implementation. This, however, can cause significant performance degradation since all reads and writes will have to be serviced by a shared cache, effectively rendering local caches useless.

4.1.3 Approach

In the basic scheme, for each cache line in the shared cache, we keep track of whether the line is exclusive, shared or read-only. Shared lines do *not require tracking of sharers*. Additionally, for private cache lines, we only maintain a pointer to the owner.

Since we do not track sharers, writes do not eagerly invalidate shared copies in other processors. On the contrary, writes are merely propagated to the shared cache in program order (thus ensuring write-write order). To save bandwidth, instead of writing the full data block to the shared cache, we merely propagate the coherence states. In the above example, the writes a_1 and a_2 are guaranteed to propagate to the shared cache in the correct order. Intuitively, the *most recent* value of any data is maintained in the shared cache.

Reads to shared cache lines are allowed to read from the local cache, up to a predefined number of accesses (potentially causing a stale value to be read), but are forced to re-request the cache line from the shared cache after exceeding an access threshold (our implementation maintains an access counter per line). This ensures that any write (used as a release) will eventually be made visible to the matching acquire, ensuring *eventual write propagation*. In the above example, this ensures that the read b_1 will eventually access the shared cache and see the update from a_2.

When a read misses in the local cache, it is forced to obtain the most recent value from the shared cache. In order to ensure the read-read order, future reads will also need to read the most recent values. To guarantee this, whenever a read misses in the local cache, we self-invalidate all shared cache lines. In the above example, whenever b_1 sees the update from a_2, self-invalidation ensures that b_2 correctly reads the value produced by a_1.

Finally, we reduce the number of self-invalidations by employing *timestamps*. Timestamps are commonly used when inferring logical ordering among distributed events

is sufficient [Lam78]. Since memory consistency talks about logical ordering among events, the use of timestamps follows naturally; indeed, timestamps have been used in the past for memory consistency implementation and verification [MB92; NN94; Pla+98; Shi+11].

More specifically, to reduce self-invalidations we employ a variant of transitive reduction [Net93]. If at a read miss, the corresponding write is determined to have happened before a previously seen write, self-invalidation is not necessary. In the example, even though b_2 reads from the shared cache, this does not cause self-invalidation.

4.2 TSO-CC: Protocol Design

This section outlines the design and implementation details of the protocol: first we present a conceptual overview, followed by the basic version of the protocol, and then proceed incrementally adding optimizations to further exploit the relaxations of TSO. We assume a local L1 cache per core and a NUCA [KBK02] architecture for the shared L2 cache.

4.2.1 Overview

To keep the TSO-CC protocol scalable, we do not want to use a full sharing vector. Thus, a major challenge is to enforce TSO without a full sharing vector, while minimizing costly invalidation messages—a consequence of which is that the resulting protocol must enforce coherence lazily.

Our basic approach is as follows. When a write retires from the write-buffer, instead of eagerly propagating it to all sharers like a conventional eager coherence protocol, we merely propagate the write to the shared cache. One way to do this is to simply write through to the shared cache. To save bandwidth, however, our protocol uses a write-back policy, in that, only state changes are propagated to the shared cache. In addition to this, by delaying subsequent writes until the previous write's state changes have been acknowledged by the shared cache, we ensure that writes are propagated to the shared cache in program order. Informally, this ensures that the "most recent" value of any address can be obtained by sending a request to the shared cache.

Consequently, one way to ensure write propagation trivially is for all reads to read from the shared cache [Pug+10]. Note that this would ensure that all reads would get the most recent value, which in turn would ensure that any write which is used as a

release (i.e. a synchronization operation) would definitely be seen by its matching acquire. However, the obvious problem with this approach is that it effectively means that shared data cannot be cached, which can affect performance significantly as we will show later with our experiments.

We ensure eventual write propagation as follows. First, let us note that ensuring write propagation means that a write is *eventually* propagated to all processors. The keyword here is *eventually*, as there is no guarantee on *when* the propagation will occur even for shared memory systems that enforce the strongest memory consistency model (SC) using eager coherence. Consequently, shared memory systems must be programmed to work correctly even in the presence of propagation delays. While this is typically accomplished by employing proper *synchronization*, unsynchronized operations are used in shared memory systems as well. For example, synchronization constructs themselves are typically constructed using unsynchronized writes (releases) and reads (acquires). The same rules apply even with unsynchronized operations. Shared memory systems using unsynchronized operations may rely on the fact that an unsynchronized write (for e.g. release) would eventually be made visible to a read (for e.g. acquire), but must be tolerant to propagation delays. In other words, the corresponding unsynchronized read (acquire) must continually read the value to see if the write has propagated. This is precisely why all acquire-like operations have a polling read to check the synchronization value [Tia+08; Xio+10]. This is our key observation.

Motivated by this observation, we use a simple scheme in which shared reads are allowed to hit in the local cache a predefined number of times, before forcing a miss and reading from the lower-level cache. This guarantees that those reads that are used as acquires will definitely see the value of the matching release, while ensuring that other shared data are allowed to be cached. It is important to note that in doing this optimization, we are not imposing any particular shared-memory programming model. Indeed, our experiments show that our system can work correctly for a wide variety of lock-based and lock-free programs.

Having guaranteed eventual write propagation, we now explain how we ensure the memory orderings guaranteed by TSO. We already explained how, by propagating writes to the shared cache in program order, we ensure the write-write ordering. Ensuring the read-read ordering implies that the second read should appear to perform after the first read. Whenever a read is forced to obtain its value from the shared cache (due to any miss, be it capacity/cold, or a shared read that exceeded the maximum allowed accesses),

and the last writer is not the requesting core, we *self-invalidate all shared cache lines* in the local cache. This ensures that future reads are forced to obtain the most recent data from the shared cache, thereby ensuring read-read ordering. The read-write order is trivially ensured as writes retire into the write-buffer only after all preceding reads complete.

4.2.2 Basic Protocol

Having explained the basic approach, we now discuss in detail our protocol. A detailed state transition table (including transient states) can be found in Appendix A. First, we start with the basic states, and explain the actions for reads, writes, and evictions.

Stable states: The basic protocol distinguishes between invalid (Invalid), private (Exclusive, Modified) and shared (Shared) cache lines, but does not require maintaining a sharing vector. Instead, in the case of private lines—state Exclusive in the L2—the protocol only maintains a pointer b.owner, tracking which core owns a line; shared lines are untracked in the L2. The L2 maintains an additional state Uncached denoting that no L1 has a copy of the cache line, but is valid in the L2.

Reads: Similar to a conventional MESI protocol, read requests (GetS) to invalid cache lines in the L2 result in Exclusive responses to L1s, which must acknowledge receipt of the cache line. If, however, a cache line is already in private state in the L2, and another core requests read access to the line, the request is forwarded to the owner. The owner will then downgrade its copy to the Shared state, forward the line to the requester and sends an acknowledgement to the L2, which will also transition the line to the Shared state. On subsequent read requests to a Shared line, the L2 immediately replies with Shared data responses, which do not require acknowledgement by L1s.

Unlike a conventional MESI protocol, Shared lines in the L1 are allowed to hit upon a read, *only* until some predefined maximum number of accesses, at which point the line has to be re-requested from the L2. This requires extra storage for the access counter b.acnt—the number of bits depend on the maximum number of L1 accesses to a Shared line allowed.

As Shared lines are untracked, each L1 that obtains the line must eventually self-invalidate it. *After any L1 miss, on the data response*, where the last writer is not the requesting core, *all Shared lines must be self-invalidated*.

Writes: Similar to a conventional MESI protocol, a write can only hit in the L1 cache if the corresponding cache line is held in either Exclusive or Modified state; transitions

from Exclusive to Modified are silent. A write misses in the L1 in any other state, causing a write request (GetX) to be sent to the L2 cache and a wait for response from the L2. Upon receipt of the response from the L2, the local cache line's state changes to Modified and the write hits in the L1, finalizing the transition with an acknowledgement to the L2. The L2 cache must reflect the cache line's state with the Exclusive state and set b.owner to the requester's id. If another core requests write access to a private line, the L2 sends an invalidation message to the owner stored in b.owner, which will then pass ownership to the core which requested write access. Since the L2 only responds to write requests if it is in a stable state, i.e. it has received the acknowledgement of the last writer, there can only be one writer at a time. This serializes all writes to the same address at the L2 cache.

Unlike a conventional MESI protocol, on a write to a Shared line, the L2 immediately responds with a data response message and transitions the line to Exclusive. Note that even if the cache line is in Shared, the L2 must send the entire line, as the requesting core may have a stale copy. On receiving the data message, the L1 transitions to Exclusive either from Invalid or Shared. Note that there may still be other copies of the line in Shared in other L1 caches, but since they will eventually re-request the line and subsequently self-invalidate all Shared lines, TSO is satisfied.

Evictions: Untracked cache lines in state Shared in the L2 are not inclusive. Therefore, on evictions from the L2, only Exclusive (as well as SharedRO with the optimization introduced in §4.2.4) evictions require sending invalidation requests to the owner; Shared lines are evicted silently from the L2. Similarly for the L1, Exclusive lines need to inform the L2, which can then transition the line to Uncached; Shared lines are evicted silently from the L1.

4.2.3 Opt. 1: Reducing Self-Invalidations

In order to satisfy the read-read ordering, in the basic protocol, all L2 accesses except to lines where b.owner is the requester, result in self-invalidation of all Shared lines. This leads to shared accesses following an acquire to miss and request the cache line from the L2, and subsequently self-invalidating all shared lines again. For example in Figure 4.1, self-invalidating all Shared lines on the acquire b_1 but also on subsequent read misses is not required. This is because the self-invalidation at b_1 is supposed to make all writes before a_2 visible. Another self-invalidation happens at b_2 to make all writes before a_1 visible. However, this is unnecessary, as the self-invalidation at b_1 (to

make all writes before a_2 visible) has already taken care of this.

To reduce unnecessary invalidations, we implement a version of the transitive reduction technique outlined in [Net93]. Each line in the L2 and L1 must be able to store a timestamp b.ts of fixed size; the size of the timestamp depends on the storage requirements, but also affects the frequency of timestamp resets, which are discussed in more detail in §4.2.5. A line's timestamp is updated on every write, and the source of the timestamp is a unique, monotonically increasing core local counter, which is incremented on every write.

Thus, to reduce invalidations, only where the requested line's timestamp is *larger than the last-seen timestamp from the writer of that line*, treat the event as a *potential acquire* and self-invalidate all Shared lines.

To maintain the list of last-seen timestamps, each core maintains a timestamp table ts_L1. The maximum possible entries per timestamp table can be less than the total number of cores, but will require an eviction policy to deal with limited capacity. The L2 responds to requests with the data, the writer b.owner and the timestamp b.ts. For those data responses where the timestamp is invalid (lines which have never been written to since the L2 obtained a copy) or there does not exist an entry in the L1's timestamp-table (never read from the writer before), it is also required to self-invalidate; this is because timestamps are not propagated to main-memory and it may be possible for the line to have been modified and then evicted from the L2.

Timestamp groups: To reduce the number of timestamp resets, it is possible to assign groups of contiguous writes the same timestamp, and increment the local timestamp-source after the maximum writes to be grouped is reached. To still maintain correctness under TSO, this changes the rule for when self-invalidation is to be performed: only where the requested line's timestamp is *larger or equal* (contrary to just larger as before) *than the last-seen timestamp from the writer of that line*, self-invalidate all Shared lines.

4.2.4 Opt. 2: Shared Read-Only Data

The basic protocol does not take into account lines which are written to very infrequently but read frequently. Another problem are lines which have no valid timestamp (due to prior L2 eviction), causing frequent mandatory self-invalidations. To resolve these issues, we add another state SharedRO for shared read-only cache lines.

A line transitions to SharedRO instead of Shared if the line is not modified by the previous Exclusive owner (this prevents Shared lines with invalid timestamps).

In addition, cache lines in the Shared state *decay* after some predefined time of not being modified, causing them to transition to SharedRO. In our implementation, we compare the difference between the shared cache line's timestamp and the writer's last-seen timestamp maintained in a table of last-seen timestamps ts_L1 in the L2 (this table is reused in §4.2.5 to deal with timestamp resets). If the difference between the line's timestamp and last-seen timestamp exceeds a predefined value, the cache line is transitioned to SharedRO.

Since on a self-invalidation, only Shared lines are invalidated, this optimization already decreases the number of self-invalidations, as SharedRO lines are excluded from invalidations. Regardless, this still poses an issue, as on every SharedRO data response, the timestamp is still invalid and will cause self-invalidations. To solve this, we introduce timestamps for SharedRO lines with the timestamp-source being the L2 itself; note that, each L2 tile will maintain its own timestamp-source. The event on which a line is assigned a timestamp is on transitions from Exclusive or Shared to SharedRO. On such transitions the L2 tile increments its timestamp-source.

Each L1 must maintain a table ts_L2 of last-seen timestamps for each L2 tile. On receiving a SharedRO data line from the L2, the following rule determines if self-invalidation should occur: if the line's timestamp is *larger than the last-seen timestamp from the L2*, self-invalidate all Shared lines.

Writes to shared read-only lines: A write request to a SharedRO line requires a broadcast to all L1s to invalidate the line. To reduce the number of required broadcast invalidation and acknowledgement messages, the b.owner entry in the L2 directory is reused as a coarse sharing vector [GWM90], where each bit represents a group of sharers. As writes to SharedRO lines should be infrequent, the impact of unnecessary SharedRO invalidation/acknowledgement messages should be small.

Evictions: Evictions of SharedRO lines require broadcasting invalidations to the sharers tracked in the coarse sharing vector, with each L1 acknowledging invalidation (even if Invalid in the L1). Evictions of SharedRO lines from L1s are therefore silent.

Timestamp groups: To reduce the number of timestamp resets, the same timestamp can be assigned to groups of SharedRO lines. In order to maintain read-read ordering, a core must self-invalidate on a read to a SharedRO line that could potentially have been modified since the last time it read same line. This can only be the case, if a line ends up in a state, after a modification, from which it can reach SharedRO again:

(1) after an L2 eviction of a dirty line, or after a GetS request to a line in Uncached which has been modified;

(2) after a line transitions to the Shared state.

It suffices to have a flag for cases (1) and (2) each to denote if the timestamp-source should be incremented on a transition event to SharedRO. All flags are reset after incrementing the timestamp-source.

4.2.5 Timestamp Resets

Since timestamps are finite, we have to deal with timestamp resets for both L1 and L2 timestamps. If the timestamp and timestamp-group size are chosen appropriately, timestamp resets should occur relatively infrequently, and does not contribute overly negative to network traffic. As such, the protocol deals with timestamp resets by requiring the node, be it L1 or L2 tile, which has to reset its timestamp-source to broadcast a timestamp reset message.

In the case where an L1 requires resetting the timestamp-source, the broadcast is sent to every other L1 and L2 tile. Upon receiving a timestamp reset message, an L1 invalidates the sender's entry in the timestamp table ts_L1. However, it is possible to have lines in the L2 where the timestamp is from a previous epoch, where each epoch is the period between timestamp resets, i.e. b.ts is larger than the current timestamp-source of the corresponding owner. The only requirement is that the L2 must respond with a timestamp that reflects the correct happens-before relation.

The solution is for each L2 tile to maintain a table of last-seen timestamps ts_L1 for every L1; the corresponding entry for a writer is updated when the L2 updates a line's timestamp upon receiving a data message. Every L2 tile's last-seen timestamp table must be able to hold as many entries as there are L1s. The L2 will assign a data response message the line's timestamp b.ts if the *last-seen timestamp from the owner is larger or equal to b.ts, the smallest valid timestamp otherwise*. Similarly for requests forwarded to an L1, only that the line's timestamp is compared against the current timestamp-source.

Upon resetting an L2 tile's timestamp, a broadcast is sent to every L1. The L1s remove the entry in ts_L2 for the sending tile. To avoid sending larger timestamps than the current timestamp-source, the same rule as for responding to lines not in SharedRO as described in the previous paragraph is applied (compare against L2 tile's current timestamp-source).

One additional case must be dealt with, such that if the smallest valid timestamp is used if a line's timestamp is from a previous epoch, it is not possible for an L1 to skip

self-invalidation due to the line's timestamp being equal to the smallest valid timestamp. To address this case, the next timestamp assigned to a line after a reset must always be larger than the smallest valid timestamp.

Handling races: As it is possible for timestamp reset messages to race with data request and response messages, the case where a data response with a timestamp from a previous epoch arrives at an L1 which already received a timestamp reset message, needs to be accounted for. Waiting for acknowledgements from all nodes having a potential entry of the resetter in a timestamp table would cause twice the network traffic on a timestamp reset and unnecessarily complicates the protocol. We introduce an *epoch-id* to be maintained per timestamp-source. The epoch-id is incremented on every timestamp reset and the new epoch-id is sent along with the timestamp reset message. It is not a problem if the epoch-id overflows, as the only requirement for the epoch-id is to be distinct from its previous value. However, we assume a bound on the time it takes for a message to be delivered, and it is not possible for the epoch-id to overflow and reach the same epoch-id value of a message in transit.

Each L1 and L2 tile maintains a table of epoch-ids for every other node: L1s maintain epoch-ids for every other L1 (epoch_ids_L1) and L2 (epoch_ids_L2) tile; L2 tiles maintain epoch-ids for all L1s. Every data message that contains a timestamp, must now also contain the epoch-id of the source of the timestamp: the owner's epoch-id for non-SharedRO lines and the L2 tile's epoch-id for SharedRO lines.

Upon receipt of a data message, the L1 compares the expected epoch-id with the data message's epoch-id; if they do not match, the same action as on a timestamp reset has to be performed, and can proceed as usual if they match.

4.2.6 Atomic Accesses and Fences

Implementing atomic read and write instructions, such as RMWs, is trivial with our proposed protocol. Similarly to a conventional MESI protocol, in our protocol an atomic instruction also issues a GetX request. Fences require unconditional self-invalidation of cache lines in the Shared state.

4.2.7 Speculative Execution

The description thus far is compatible with a core with a FIFO write-buffer, but without *speculative load execution* [GGH91]. We will assume an implementation with load speculation implemented via a load-buffer as discussed in [GGH91]. With a load-buffer,

the coherence protocol must forward invalidations of cache lines, so that the load-buffer can initiate a pipeline squash on incorrect speculation.

Similarly to other conventional eager protocols, any invalidation (including self-invalidations) must be forwarded to the load-buffer. However, a corner case exists where a self-invalidation occurs and another line, due to a speculated read, is still in a transient state—state WaitS after a read (see Appendix A).

Consider the pattern in Figure 4.2. Assume the load-buffer issues the read for (2b) first (transition to WaitS), the L2 cache responds with the initial data but the response message remains in transit. Next, (1a) and (1b) are performed (and committed), and then (2a) is issued and receives the value produced by (1b). With the protocol description thus far, the acquire at (2a) causes self-invalidation of only Shared lines. However, the response for (2b) arrives with stale data, which would cause a TSO violation.

init: x = 0, y = 0	
Thread 1	**Thread 2**
(1a) x ← 1	(2a) r1 ← y
(1b) y ← 1	(2b) r2 ← x

Figure 4.2: Message passing pattern.

The solution is, that upon self-invalidation, for all lines in WaitS state, an invalidation must also be forwarded to a load-buffer. Then, to avoid a retry of a squashed load still hitting on a stale cache line, we propose adding an additional bit of information to a read request from the load-buffer to denote a retry. In case of a read+retry, the protocol forces a miss in the Shared state only. This option offers potentially higher performance, as the load-buffer has more information about which instructions are potentially violated or not, and could still hit (more than once) in a stale cache line if no violation is detected.

4.2.8 Storage Requirements and Organization

Table 4.1 shows a detailed breakdown of storage requirements for a TSO-CC implementation, referring to literals introduced in §4.2. Per cache line storage requirements has the most significant impact, which scales logarithmically with increasing number of cores (see §4.4, Figure 4.3).

While we chose a simple sparse directory embedded in the L2 cache for our evaluation (Figure 4.3), our protocol is independent of a particular directory organization.

Table 4.1: TSO-CC specific storage requirements.

L1	**Per node:** • Current timestamp, B_{ts} bits • Write-group counter, $B_{write-group}$ bits • Current epoch-id, $B_{epoch-id}$ bits • Timestamp-table ts_L1$[n]$, $n \leq Count_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = Count_{L1}$ entries Only required if SharedRO opt. (§4.2.4) is used: • Timestamp-table ts_L2$[n]$, $n \leq Count_{L2-tiles}$ entries • Epoch-ids epoch_ids_L2$[n]$, $n = Count_{L2-tiles}$ entries **Per line b:** • Number of accesses b.acnt, B_{maxacc} bits • Last-written timestamp b.ts, B_{ts} bits
L2	**Per tile:** • Last-seen timestamp-table ts_L1, $n = Count_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = Count_{L1}$ entries Only required if SharedRO opt. (§4.2.4) is used: • Current timestamp, B_{ts} bits • Current epoch-id, $B_{epoch-id}$ bits • Increment-timestamp-flags, 2 bits **Per line b:** • Timestamp b.ts, B_{ts} bits • Owner (Exclusive), last-writer (Shared), coarse vector (SharedRO) as b.owner, $\lceil log(C_{L1}) \rceil$ bits

It is possible to further optimize our overall scheme by using directory organization approaches such as in [Fer+11; SK12]; however, this is beyond the scope of this thesis. Also note that we do not require inclusivity for Shared lines, alleviating some of the set conflict issues associated with the chosen organization.

4.3 Proof of Correctness

This section aims at providing a proof sketch that TSO-CC satisfies TSO. While designing a protocol, the very first steps taken typically involve coming up with an abstract machine that illustrates the designer's intent and refining this abstract machine until a full protocol materializes. The design process for TSO-CC was no different, in that we initially reasoned (informally) about a very abstract machine satisfying TSO, and refined it towards a complete implementation. Even the preceding description is necessarily more abstract than the full protocol with all transient states presented in Appendix A. The proof strategy taken here is a formalization of our informal reasoning approach, which ultimately leads to a simpler proof (rather than a direct proof against axiomatic TSO).

More specifically, the general proof strategy is to prove all states of TSO-CC are *weakly simulated* by some states of a more abstract *operational model* which has been proved to satisfy TSO. Weak simulation (unlike strong simulation) admits proofs where either model can have numerous internal transitions that are not observable (unlabeled) and there may not be a direct correspondence between internal transitions. In the following we first define labeled transition systems [Plo81], which are used to describe the operational semantics of the abstract machine, followed by a summary of the definitions from [Mil99] required for the proof.

Definition 4.1 (Labeled Transition System)**.** A labeled transition system (LTS) is a structure $\langle \mathcal{S}, \mathcal{L}, \longrightarrow \rangle$ where $\mathcal{S}$ is a set of states, $\mathcal{L}$ is a set of labels (or actions) and $\longrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the transition relation. We use the shorthand for a transition $s \xrightarrow{l} s' \triangleq (s,l,s') \in \longrightarrow$, where $s, s' \in \mathcal{S}$ and $l \in \mathcal{L}$.

Definition 4.2 (Derived transition relations)**.** The expression $s \Longrightarrow s'$ denotes the sequence of zero or more transitions $s \longrightarrow \cdots \longrightarrow s'$, i.e. the reflexive-transitive closure of the transition relation: $\Longrightarrow \triangleq \longrightarrow^*$. Let $ls = l_0 \cdots l_n$ be a sequence of actions; then $s \xRightarrow{ls} s'$ denotes $s \Longrightarrow \cdots \xrightarrow{l_0} \cdots \xrightarrow{l_n} \cdots \Longrightarrow s'$, i.e. the trace of transitions that gives rise to observable sequence of actions ls: $\xRightarrow{ls} \triangleq (\Longrightarrow; \xrightarrow{l_0}; \Longrightarrow; \cdots; \Longrightarrow; \xrightarrow{l_n}; \Longrightarrow)$,

where $\xrightarrow{\lambda} \triangleq \{(s,l,s') \mid (s,l,s') \in \longrightarrow \wedge\, l = \lambda\}$.

Definition 4.3 (Weak simulation)**.** Let $\mathcal{R}$ be a binary relation over $\mathcal{S}$. $\mathcal{R}$ is a weak simulation on an LTS, if for all $p\mathcal{R}q$ (p simulated by q),

$$\text{if } p \stackrel{ls}{\Longrightarrow} p' \text{ then } \exists q' \in \mathcal{S}.\ q \stackrel{ls}{\Longrightarrow} q' \wedge p'\mathcal{R}q'$$

that is, all visible actions that can be made from p to p' can be matched by some visible action from q to q', and p' is simulated by q'.

Proposition 4.1 (Weak simulation proof rule)**.** *To prove that $\mathcal{R}$ is a weak simulation, it is sufficient to show that, for all $p\mathcal{R}q$*

- *if $p \longrightarrow p'$ then $\exists q' \in \mathcal{S}.\ q \Longrightarrow q' \wedge p'\mathcal{R}q'$*
- *if $p \xrightarrow{\lambda} p'$ then $\exists q' \in \mathcal{S}.\ q \stackrel{\lambda}{\Longrightarrow} q' \wedge p'\mathcal{R}q'$*

To prove the correctness of a *coherence protocol against some abstract operational model capturing memory consistency behavior, we need to show that every visible action (reads and writes) by the protocol can be matched by the more abstract operational model*. That is, the states of the protocol are simulated by the abstract operational model.

4.3.1 Abstract TSO Load-Buffering Machine

This section introduces the abstract TSO load-buffering machine (TSO-LB-ABS). Although there exist operational models of TSO, e.g. [OSS09], these models use a store-buffering model, and a mapping to the state-space of a coherence protocol is less obvious (as is later needed by the simulation proof). As the lazy coherence protocols proposed effectively follow a load-buffering approach, with loads, viz. reads, hitting on a "buffered" or stale value, a different model is proposed. Write misses on the other hand are not taken out of the critical path by the protocol itself (but may be independently by a core entering these into a store-buffer).

Definition 4.4 (Abstract TSO machine TSO-LB-ABS)**.** We will define the LTS for TSO-LB-ABS inductively, over the following inference rules, where:

- P is a set of processors, of which $p \in P$;
- A is a set of addresses (memory locations), of which $a \in A$;

- V is a set of data values, of which $v \in V$;
- $\texttt{local} : (\mathcal{S} \times P \times A) \to V$ is a function where $\texttt{local}(s,p,a)$ is the value at address a in the local store of p in state s;
- $\texttt{global} : (\mathcal{S} \times A) \to V$ is a function where $\texttt{global}(s,a)$ is the value at address a in the global store in state s.

$$\frac{\texttt{local}(s,p,a) = v}{s \xrightarrow{\mathsf{Read}(p,a,v)} s} \textsc{Read}$$

$$\frac{\begin{array}{l}\texttt{global}(s,a) = v \\ s \xrightarrow{\mathsf{Write}(p,a,v,v')} s' \\ \wedge \quad \texttt{local}(s',p,a) \mapsto v' \\ \wedge \quad \texttt{global}(s',a) \mapsto v'\end{array}}{\top} \textsc{Write}$$

$$\frac{\begin{array}{l}\top \\ s \xrightarrow{\mathsf{Prop}(p)} s' \\ \wedge \quad \forall a \in A.\ \texttt{local}(s',p,a) \mapsto \texttt{global}(s,a)\end{array}}{\top} \textsc{Propagate}$$

$$\frac{\top}{s \longrightarrow s} \textsc{Stutter}$$

In the following we wish to prove that TSO-LB-ABS satisfies TSO. We prove the machine against the axiomatic TSO memory consistency model that is a result of instantiating the framework in §2.2 with Definition 2.4. Here, we will only prove that all behaviors of TSO-LB-ABS satisfy TSO, but not that all behaviors admitted under TSO are allowed by TSO-LB-ABS; for all intents and purposes, TSO-LB-ABS is as strict or stricter than TSO, which is sufficient to prove correctness of TSO-CC.

Definition 4.5 (Set of execution witnesses of TSO-LB-ABS)**.** Let E be the set of execution witnesses obtained by constructing the relations po, rf and co (see Definition 2.2) over the trace of labels $l \in L$ from the initial states I, where

$$I \triangleq \{s \mid \forall p \in P.\ \forall a \in A.\ \texttt{local}(s,p,a) = \texttt{global}(s,a)\}$$

In order to prove the final Theorem 4.1—that TSO-LB-ABS satisfies axiomatic TSO—we must prove all four constraints defined in §2.2.4 are satisfied by the executions E of TSO-LB-ABS. First, however, we will prove some helpful intermediate lemmas.

Lemma 4.1 (All co are well-formed)**.** *Writes to the same memory location are totally ordered in* co *for all execution witnesses E of* TSO-LB-ABS.

Proof. This is trivially true in the initial states I, as $\mathsf{co} = \emptyset$. Since writes only happen in one rule (WRITE), with action $\mathsf{Write}(p,a,v,v')$ where v' overwrites the previous value v (of event w), we only need to show that no two writes can overwrite the value of the unique event w—for the purpose of this proof, this assumes unique write values to establish this mapping. Since $\texttt{global}(s,a) = v$, the value to be overwritten by v', in s and $\texttt{global}(s',a) = v'$ in the next state s', it is impossible for any two writes to overwrite the same value. Therefore, co must be a total order over the writes to location a. □

Lemma 4.2. *Let T be the set of states in a trace from s_0 to s_n, such that*

$$\cdots \xrightarrow{\mathit{Prop}(p)} s_0 \cdots s_n \xrightarrow{\mathit{Write}(p,a,v',v)} s'$$

Then for all states $s_i \in T$, the local store of p is equal to the global store in s_i or some preceding state in the trace.

Proof. By rule induction on TSO-LB-ABS.

Case READ: The local store is not updated.

Case PROPAGATE: Suppose that the hypothesis is true in s, then by definition $\forall a \in A.\ \texttt{local}(s,p,a) = \texttt{global}(s',a)$, and therefore holds in s'.

□

Lemma 4.3. *Values written by p are immediately visible to p, i.e.*

$$s \xrightarrow{\mathit{Write}(p,a,v,v')} s' \cdots s'' \xrightarrow{\mathit{Read}(p,a,v')} \cdots$$

without a propagation transition between s' and s''.

Proof. Since writes to a in rule WRITE also update the memory location a in the local store of p, any following reads (by definition only from the local store) by p must read the value written by p to a (until propagation). □

Lemma 4.4 (TSO-LB-ABS satisfies SC PER LOCATION)**.** *All execution witnesses E of* TSO-LB-ABS *satisfy* SC PER LOCATION *(Definition 2.5).*

Proof. This is true if the order of operations for one memory location never appear to be reordered. That is, after a $\mathsf{Read}(p,a,v)$ or $\mathsf{Write}(p,a,v',v)$, it is impossible to read a value older than v. We can show this by Lemmas 4.2 and 4.3, and assert that writes to the same location are totally ordered by Lemma 4.1. □

Lemma 4.5 (TSO-LB-ABS satisfies NO THIN AIR)**.** *All execution witnesses E of* TSO-LB-ABS *satisfy* NO THIN AIR *(Definition 2.6) instantiated with TSO (Definition 2.4).*

Proof. This is true if we can show that reads that read a value v, cannot be ordered via ppo (fences are always empty for E of TSO-LB-ABS) and rf before the producing write of v. That is, there does not exist $r_0 \overset{\text{ppo}}{\rightarrow} w_0 \wedge r_1 \overset{\text{ppo}}{\rightarrow} w_1 \wedge w_1 \overset{\text{rfe}}{\rightarrow} r_0 \wedge w_0 \overset{\text{rfe}}{\rightarrow} r_1$. Since in TSO-LB-ABS write actions occur in the same transition as its effect on the memory system, it is impossible for a read to observe the write of a future transition. □

Lemma 4.6 (TSO-LB-ABS satisfies OBSERVATION with TSO)**.** *All execution witnesses E of* TSO-LB-ABS *satisfy* OBSERVATION *(Definition 2.7) instantiated with TSO (Definition 2.4).*

Proof. We must show that writes that are ordered via prop, are observed in that order. It suffices to show that the local store in p always contains a snapshot of the global store (modulo writes by p, as WR in program order is relaxed), as this implies it is impossible to observe writes out of order or reorder reads. By Lemma 4.2 this is true. □

Lemma 4.7 (TSO-LB-ABS satisfies PROPAGATION with TSO)**.** *All execution witnesses E of* TSO-LB-ABS *satisfy* PROPAGATION *(Definition 2.8) instantiated with TSO (Definition 2.4).*

Proof. We must show all threads observe writes of other (rfe only) threads in the same order, since prop includes ppo (asserting write/multi-copy atomicity). Again, it suffices to show that the local store in p always contains a snapshot of the global store (modulo writes by p) via Lemma 4.2. □

Theorem 4.1 (TSO-LB-ABS satisfies TSO)**.** *All execution witnesses E of* TSO-LB-ABS *satisfy the TSO axiomatic memory consistency model (instantiated with Definition 2.4).*

Proof. We can prove that TSO-LB-ABS satisfied by having shown that each axiom holds. By Lemmas 4.4 (proving SC PER LOCATION), 4.5 (proving NO THIN AIR), 4.6 (proving OBSERVATION) and 4.7 (proving PROPAGATION) we conclude that TSO-LB-ABS satisfies TSO. □

4.3.2 Sketch for Unoptimized Protocol

In the following, we present a proof sketch that the TSO-CC-basic (TSO-CC without optimizations) protocol satisfies the TSO consistency model by showing it is simulated by TSO-LB-ABS. The fully optimized protocols should immediately follow from the unoptimized protocol, but requires elaboration on the particular optimizations, which have sufficiently been argued to be correct in §4.1.3.

Protocol-intrinsic issues, in particular, race conditions, deadlock and livelock freedom [MS91], are not discussed; it is beyond the scope what can comfortably be done manually and should be automated via e.g. a model checker. Indeed, even formal methods based approaches such as model checking easily reach their limits when dealing with cache coherence protocols [ASL03]. We have verified with the help of a model checker that the protocol with all optimizations given in Appendix A is free from race conditions and deadlocks (see §4.4.4).

Assumptions: From a memory consistency perspective, only reads and writes are observable actions, and hence we will only consider transitions labeled with Read$(\ldots)$ and Write$(\ldots)$ as observable; all others are considered internal and treated as unlabeled. Furthermore, in the following we will assume that the core model issuing read/write requests to the cache controllers implementing TSO-CC-basic is in-order; a read/write is completed—that is a labeled Read$(\ldots)$ or Write$(\ldots)$ transition—when the cache controller unblocks the in-order core with the expected response.

Assuming a composed LTS with TSO-CC-basic and TSO-LB-ABS, we first define the state relation between states from TSO-CC-basic and TSO-LB-ABS.

Definition 4.6 (Related states)**.** A state p from TSO-CC-basic is related to a state q from TSO-LB-ABS iff

- For all valid and non-Exclusive addresses a in the L2 with value v, TSO-LB-ABS contains a matching address in the global store s.t. $\texttt{global}(q,a) = v$.
- For all invalid addresses a in the L2 and a value in main memory of v, TSO-LB-ABS contains a matching address in the global store s.t. $\texttt{global}(q,a) = v$.
- For all L1s p, all addresses a valid in the L1 with value v, TSO-LB-ABS contains a matching address in the respective local store s.t. $\texttt{local}(q,p,a) = v$.

Lemma 4.8 (TSO-CC-basic is weakly simulated by TSO-LB-ABS)**.** *TSO-CC-basic is weakly simulated by* TSO-LB-ABS *iff*

- *For all observable read/write transitions from TSO-CC-basic states p to p', there exists a sequence of transitions from* TSO-LB-ABS *state q to q' with the same observable action, and p' and q' are still related as per Definition 4.6.*

- *For all non-observable (evictions) transitions from TSO-CC-basic states p to p', there exists a sequence of transitions from* TSO-LB-ABS *state q to q' with no observable action, and p' and q' are still related as per Definition 4.6.*

Proof (sketch). The proof is by rule induction. We will not enumerate every possible rule, but instead argue that there is symmetry between most rules as follows.

First, we argue that the L2 and main memory always match the TSO-LB-ABS global store: as there can only be one writer at a time, and since upon transitioning away from Exclusive state a cache line must always be propagated to the L2 before proceeding, the L2 always matches the global store; upon eviction from L2, the value is propagated to main memory.

Next, we argue that in the absence of L1 cache misses, the states are always related. It is obvious to see that, without misses, no propagation of writes occurs and the L1 always matches the TSO-LB-ABS local store.

Upon misses, however, we must show that the value obtained from the L2 is correct and the L1 still matches the TSO-LB-ABS local store. Correct data is always received from the L2, as we have established that the L2 always matches the TSO-LB-ABS global store. For forwarded requests, the last write is by the forwarder, thus matches TSO-LB-ABS.

As the only data in the L1 that may become stale with respect to the L2 is Shared data, we must show that after any miss, the L1 still matches the TSO-LB-ABS local store after completion of the request that initiated the miss. Due to self-invalidating all other Shared cache lines after a miss observing a write by another L1, there is no other valid Shared line left in the L1, and thus the L1 will match the TSO-LB-ABS local store after a matching transition generated by the PROPAGATE rule.

□

As TSO-LB-ABS makes no guarantees that reads eventually see updated data (the propagation rule is independent), we have to separately prove that TSO-CC propagates writes eventually.

Lemma 4.9 (Eventual write propagation)**.** *Eventually, all writes propagate to all other private caches (assuming polling reads [Tia+08; Xio+10]).*

Proof (sketch). Given a write to some address, all L1s holding the line in non-Shared state will no longer be caching the line after completion of the write. Thus write propagation is immediate, upon following reads to the cache line. Only cache lines in the Shared state may retain the line, but are forced to miss eventually (assuming polling reads to the line): the cache line's accesses counter will reach the maximum permitted accesses and is forced to miss and obtain the most up-to-date copy of the line. □

Theorem 4.2 (TSO-CC-basic satisfies TSO)**.** *TSO-CC-basic satisfies TSO, that is TSO-CC-basic is weakly simulated by* TSO-LB-ABS *and eventually all writes propagate.*

Proof. By Lemmas 4.8 (weak simulation) and 4.9 (eventual write propagation). □

4.4 Evaluation Methodology

This section provides an overview of our evaluation methodology used in obtaining the performance results (§4.5). We also discuss storage overheads of the protocol configurations used in §4.4.3.

4.4.1 Simulation Environment

For the evaluation of TSO-CC, we use the Gem5 simulator [Bin+11] in Ruby *full-system* mode. GARNET [Aga+09] is used to model the on-chip interconnect. The ISA used is x86-64, as it is the most widely used architecture that assumes a variant of TSO. The processor model used for each CMP core is a simple out-of-order processor. Table 4.2 shows the key-parameters of the system.

As TSO-CC explicitly allows accesses to stale data, this needs to be reflected in the *functional execution* (not just the timing) of the simulated execution traces. We added support to the simulator to functionally reflect cache hits to stale data, as the stock version of Gem5 in full-system mode would assume the caches to always be coherent otherwise.

4.4.2 Workloads

Table 4.3 shows the benchmarks we have selected from the PARSEC [Bie+08], SPLASH-2 [Woo+95] and STAMP [Min+08] benchmarks suites. The STAMP benchmark suite has been chosen to evaluate transactional synchronization compared to the more

Table 4.2: System parameters.

Core-count & frequency	32 (out-of-order) @ 2GHz
Write buffer entries ROB entries	32, FIFO 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	1MB×32 tiles, 64B lines, 16-way 30 to 80 cycles
Memory Memory hit latency	2GB 120 to 230 cycles
On-chip network	2D Mesh, 4 rows, 16B flits
Kernel	Linux 2.6.32.60

traditional approach from PARSEC and SPLASH-2; the STM algorithm used is *NOrec* [DSS10].

Note that in the evaluated results, we include two versions of *lu*, with and without the use of contiguous block allocation. The version which makes use of contiguous block allocation avoids false sharing, whereas the non-contiguous version does not. Both version are included to show the effect of false-sharing, as previous works have shown lazy protocols to perform better in the presence of false-sharing [Dub+91].

All selected workloads correctly run to completion with both the MESI baseline and our configurations. It should also be emphasized that all presented program codes run unmodified (including the Linux kernel [Lin]) with the TSO-CC protocol.

4.4.3 Protocol Configurations and Storage Overheads

In order to evaluate our claims, we compare against the existing Gem5 implementation of the MESI baseline directory protocol (§3.3.1). The MESI baseline directory protocol implementation part of Gem5 provides a fair baseline as it is also used by numerous related works, and its history can be traced back to the original Wisconsin GEMS simulation toolset [Mar+05], and is similar to protocols found in commercial processors. To assess the performance of TSO-CC, we have selected a range of configurations to show the impact of varying the timestamp and write-group size parameters.

We start out with a basic selection of parameters which we derived from a limited design-space exploration. We have determined 4 bits for the per-line access counter to

Table 4.3: Benchmarks and their input parameters.

PARSEC	blackscholes	simmedium
	canneal	simsmall
	dedup	simsmall
	fluidanimate	simsmall
	x264	simsmall
SPLASH-2	fft	64K points
	lu	512×512 matrix, 16×16 blocks
	radix	256K, radix 1024
	raytrace	car
	water-nsquared	512 molecules
STAMP	bayes	-v32 -r1024 -n2 -p20 -i2 -e2
	genome	-g512 -s32 -n32768
	intruder	-a10 -l4 -n2048 -s1
	ssca2	-s13 -i1.0 -u1.0 -l3 -p3
	vacation	-n4 -q60 -u90 -r16384 -t4096

be a good balance between average performance and storage-requirements, since higher values do not yield a consistent improvement in performance; this allows at most 16 consecutive L1 hits for Shared lines.

Furthermore, in all cases the shared read-only optimization as described in §4.2.4 contributes a significant improvement: average execution time is reduced by more than 35% and average on-chip network traffic by more than 75%. Therefore, we only consider configurations with the shared read-only optimization. The decay time (for transitioning Shared to SharedRO) is set to a fixed number of writes, as reflected by the timestamp (taking into account write-group size); we have determined 256 writes to be a good value.

We note however that different workloads will exhibit different data usage patterns, and the decay times determining when shared read-write data is re-classified as shared read-only may not always be the same; crucially, the use of sophisticated predictors may be required due to the variability in decay times even within workloads as proposed by Hu, Martonosi, and Kaxiras [HMK02]. For the workloads we study, the determined decay time performs well on average, but a constant value has its limitations beyond a fixed set of workloads.

Below we consider the following configurations: CC-shared-to-L2, TSO-CC-4-basic, TSO-CC-4-noreset, TSO-CC-4-12-3, TSO-CC-4-12-0, TSO-CC-4-9-3. From the parameter names introduced in Table 4.1, the naming convention used is TSO-CC-

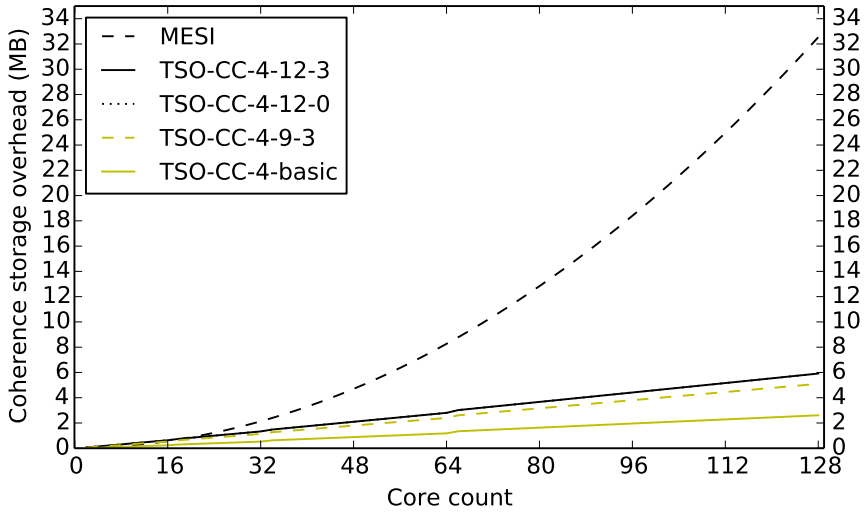


Figure 4.3: Storage overhead with all optimizations enabled, 1MB per L2 tile, and as many tiles as cores; the timestamp-table sizes match the number of cores and L2 tiles; $B_{epoch-id} = 3$ bits per epoch-id.

B_{maxacc}-B_{ts}-$B_{write-group}$.

CC-shared-to-L2: A simple protocol that removes the sharing list, but as a result, reads to Shared lines always miss in the L1 and must request the data from the L2. The base protocol implementation is the same as TSO-CC, and also includes the shared read-only optimization (without the ability to decay Shared lines, due to no timestamps). With a system configuration as in Table 4.2, CC-shared-to-L2 reduces coherence storage requirements by 76% compared to the MESI baseline.

TSO-CC-4-basic: An implementation of the protocol as described in §4.2.2 with the shared read-only optimization. Over CC-shared-to-L2, TSO-CC-4-basic only requires additional storage for the per L1 line accesses counter. TSO-CC-4-basic reduces storage requirements by 75% for 32 cores.

TSO-CC-4-noreset: Adds the optimization described in §4.2.3, but assumes infinite timestamps[1] to eliminate timestamp reset events, and increments the timestamp-source on every write, i.e. write-group size of 1. This configuration is expected to result in the lowest self-invalidation count, as timestamp-resets also affect invalidations negatively.

To assess the effect of the timestamp and the write-group sizes using realistic (feasible to implement) storage requirements, the following configurations have been selected.

[1]The simulator implementation uses 31 bit timestamps, which is more than sufficient to eliminate timestamp reset events for the chosen workloads.

TSO-CC-4-12-3: From evaluating a range of realistic protocols, this particular configuration results in the best trade-off between *storage*, and performance (in terms of *execution times* and *network traffic*). In this configuration 12 bits are used for timestamps and the write-group size is 8 (3 bits extra storage required per L1). The storage reduction over the MESI baseline is 38% for 32 cores.

TSO-CC-4-12-0: In this configuration the write-group size is decreased to 1, to show the effect of varying the write-group size. The reduction in storage overhead over the MESI baseline is 38% for 32 cores.

TSO-CC-4-9-3: This configuration was chosen to show the effect of varying the timestamp bits, while keeping the write-group size the same. The timestamp size is reduced to 9 bits, and write-group size is kept at 8. On-chip coherence storage overhead is reduced by 47% over the MESI baseline for 32 cores. Note that timestamps reset after the same number of writes as TSO-CC-4-12-0, but 8 times as often as TSO-CC-4-12-3.

Figure 4.3 shows a comparison of the extra coherence storage requirements between the MESI baseline, TSO-CC-4-12-3, TSO-CC-4-12-0, TSO-CC-4-9-3 and TSO-CC-4-basic for core counts up to 128. The best case realistic configuration TSO-CC-4-12-3 reduces on-chip storage requirements by 82% over the MESI baseline at 128 cores.

4.4.4 Verification

Initially, to check the protocol implementation for adherence to the consistency model, a set of litmus tests were chosen to be run in the full-system simulator. The diy [Alg+11] tool was used to generate litmus tests for TSO according to [OSS09]. This was invaluable in finding some of the more subtle issues in the implementation of the protocol. According to the litmus tests, each configuration of the protocol satisfies TSO. In addition, we model checked the protocol for absence of race conditions and deadlocks using Murφ [Dil96].

Furthermore, the verification challenges led to the development of McVerSi (Chapter 6). McVerSi is a framework for rigorous simulation-based memory consistency verification (of a full-system), where TSO-CC is used as a case study. It should be noted that memory consistency verification of conventional protocols and their interaction with other components (e.g. pipeline) is also addressed by McVerSi, and its importance highlighted by the fact that we discovered new bugs in Gem5's MESI protocol implementation. The presented TSO-CC protocol is, according to the McVerSi case study, correct.

Independently, Manerkar et al. [Man+15] use TSO-CC as a case study for CCICheck. CCICheck uses abstract axiomatic models of pipeline and memory system, and exhaustively verifies that a set of litmus tests is not violated. In [Man+15] the authors implement TSO-CC-basic and verify that TSO is satisfied. Given these various verification efforts, we can conclude with a high level of confidence that TSO-CC satisfies TSO.

4.5 Experimental Results

This section highlights the simulation results, and additionally gives insight into how execution times and network traffic are affected by some of the secondary properties (timestamp-resets, self-invalidations).

In the following we compare the performance of TSO-CC with the MESI baseline protocol. Figure 4.4 shows normalized (w.r.t. MESI baseline protocol) execution times and Figure 4.5 shows normalized network traffic (total flits) for all chosen benchmarks and configurations. For all TSO-CC configurations, we determine additional network traffic due to SharedRO-invalidations to be insignificant compared to all other traffic, as writes to SharedRO are too infrequent to be accounted for in Figure 4.6.

CC-shared-to-L2: We begin with showing how the naïve implementation without a sharing vector performs. On average, CC-shared-to-L2 has a slowdown of 14% over the MESI baseline; the best case, *fft*, performs 14% faster than the baseline, while the worst case has a slowdown of 84% for *lu (cont.)*. Network traffic is more sensitive, with an average increase of 137%. CC-shared-to-L2 performs poorly in cases with frequent shared misses, as seen in Figure 4.6, but much better in cases with a majority of private accesses and most shared reads are to shared read-only lines, as Figure 4.7 shows.

TSO-CC-4-basic: Compared to the baseline, TSO-CC-4-basic is 4% slower; the patterns observed are similar to CC-shared-to-L2. The best case speedup is 5% for *ssca2*, and worst case slowdown is 29% for *blackscholes*. Allowing read hits to Shared lines until the next L2 access improves execution time compared to CC-shared-to-L2 by 9%, and network traffic by 30% on average. Since the transitive reduction optimization is not used, most L1 misses cause self-invalidation as confirmed by Figure 4.8; on average 40% of read misses cause self-invalidation.

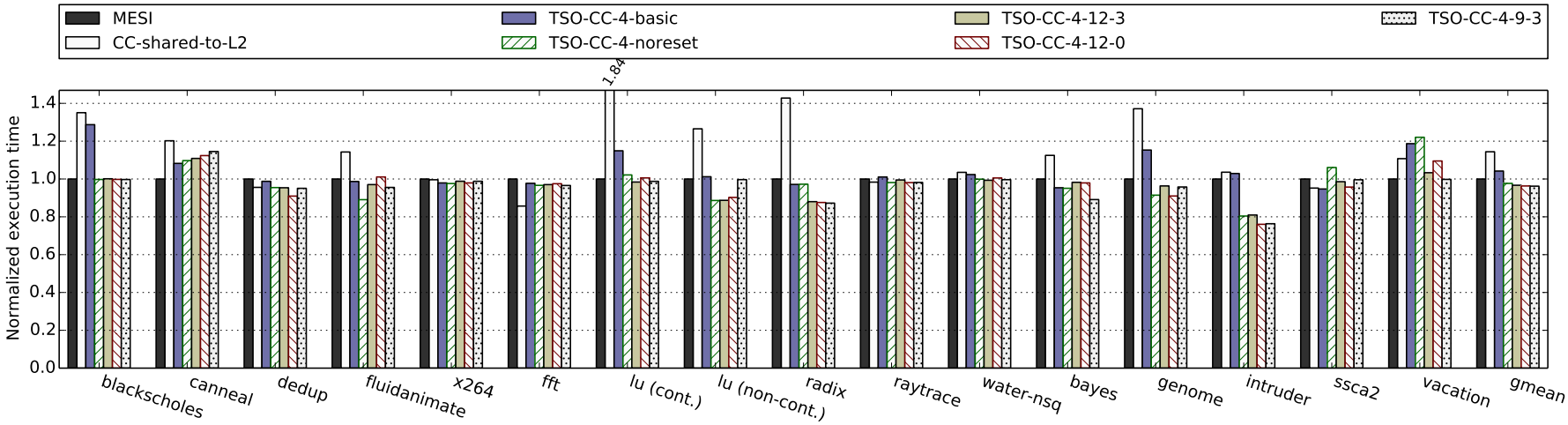


Figure 4.4: Execution times, normalized against the MESI baseline protocol.

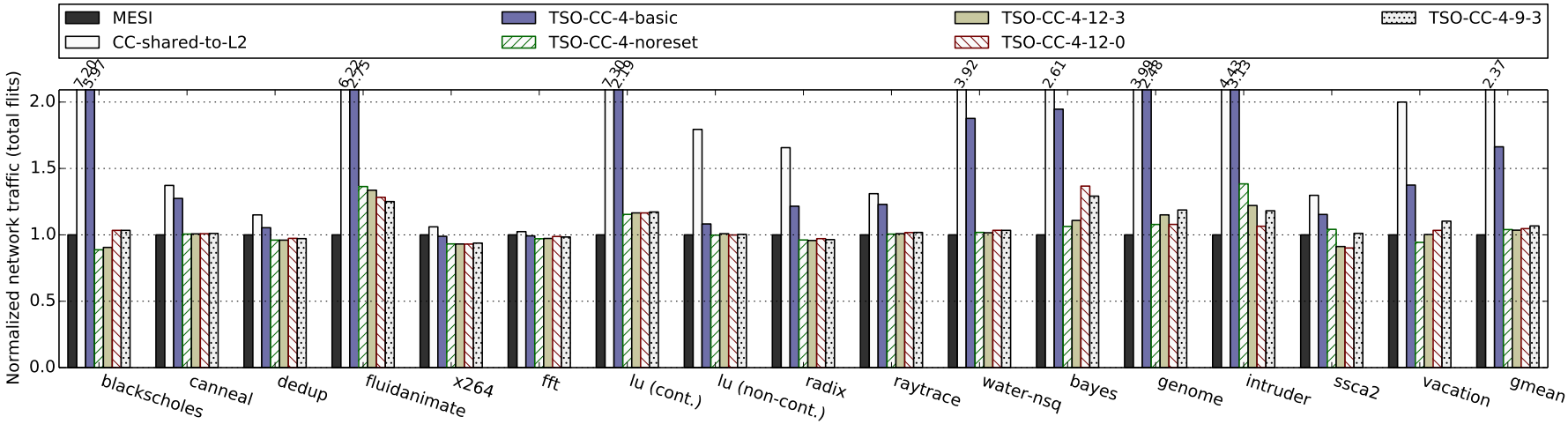


Figure 4.5: Network traffic (total flits), normalized against the MESI baseline protocol.

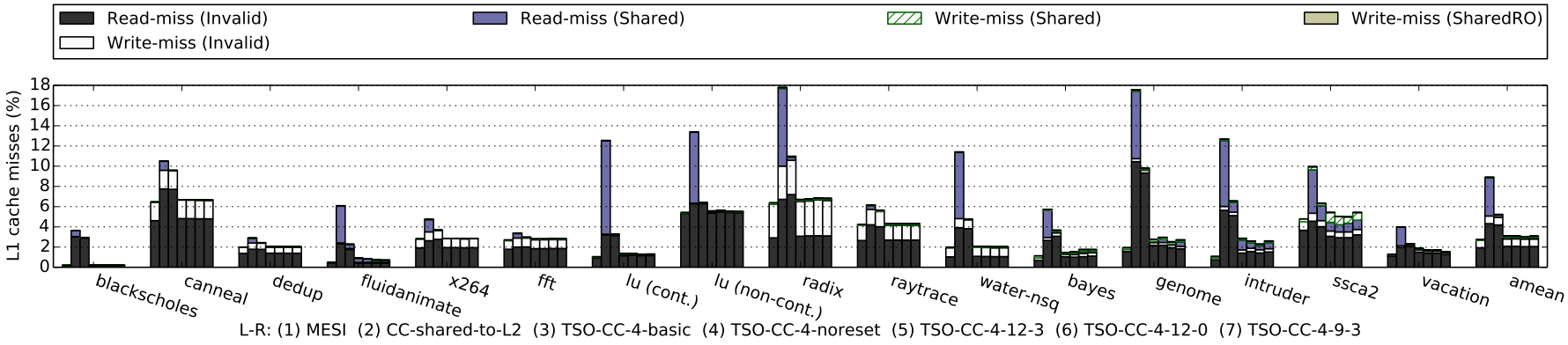


Figure 4.6: Detailed breakdown of L1 cache misses by Invalid, Shared and SharedRO states.

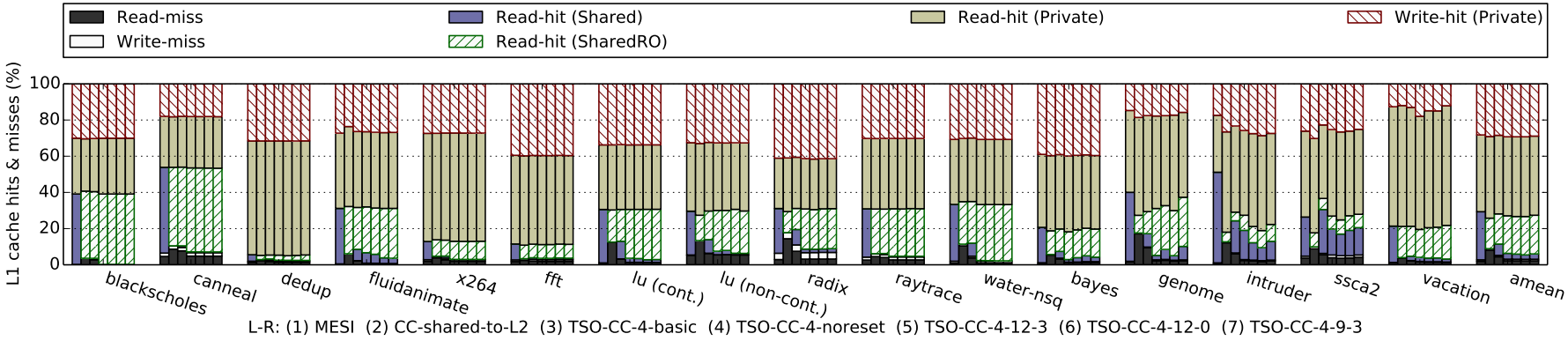


Figure 4.7: L1 cache hits and misses; hits split up by Shared, SharedRO and private (Exclusive, Modified) states.

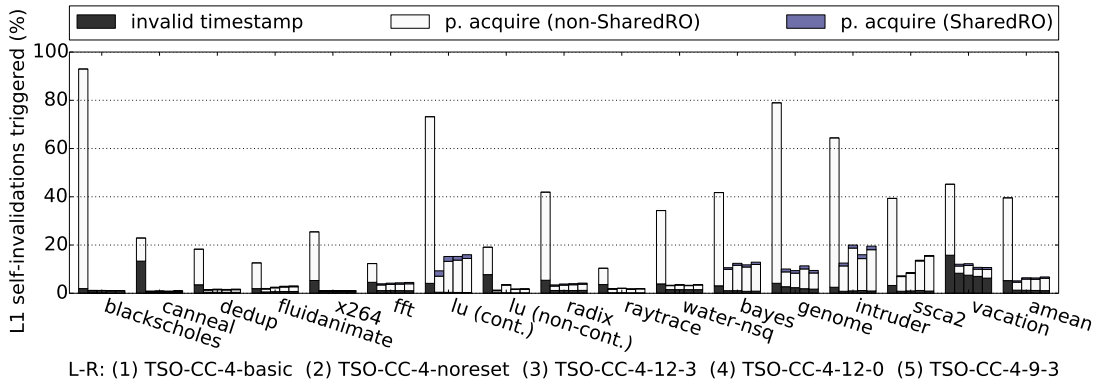


Figure 4.8: Percentage of L1 self-invalidation events triggered by data response messages.

TSO-CC-4-noreset: The ideal case TSO-CC-4-noreset shows an average of 2% improvement in execution time over the baseline; best case speedup of 20% for *intruder*, worst case slowdown of 22% for *vacation*. On average, self-invalidations—potential acquires detected as seen in Figure 4.8—are reduced by 87%, directly resulting in a speedup of 6% over TSO-CC-4-basic. Overall, TSO-CC-4-noreset requires 4% more on-chip network traffic compared to the baseline, an improvement of 37% over TSO-CC-4-basic.

TSO-CC-4-12-3: The overall best realistic configuration is on average 3% faster than the MESI baseline protocol. The best case speedup is 19% for *intruder*, and worst case slowdown is 10% for *canneal*. This configuration performs as well as TSO-CC-4-noreset (the ideal case), despite the fact that self-invalidations have increased by 25%. Over TSO-CC-4-basic, average execution time improves by 7%, as a result of reducing self-invalidations by 84%. The average network-traffic from TSO-CC-4-noreset (no timestamp resets) to TSO-CC-4-12-3 does not increase, which indicates that timestamp-reset broadcasts are insignificant compared to all other on-chip network traffic.

There are two primary reasons as to why TSO-CC-4-12-3 outperforms the MESI baseline. First, our protocol has the added benefit of reduced negative effects from false sharing, as has been shown to hold for lazy coherence protocols in general [Dub+91]. This is because shared lines are not invalidated upon another core requesting write access, and reads can continue to hit in the L1 until self-invalidated. This can be observed when comparing the two versions of *lu*. The version which does not eliminate false-sharing *(non-cont.)* performs significantly better with TSO-CC-4-12-3 compared to the MESI baseline, whereas the version where the programmer explicitly eliminates

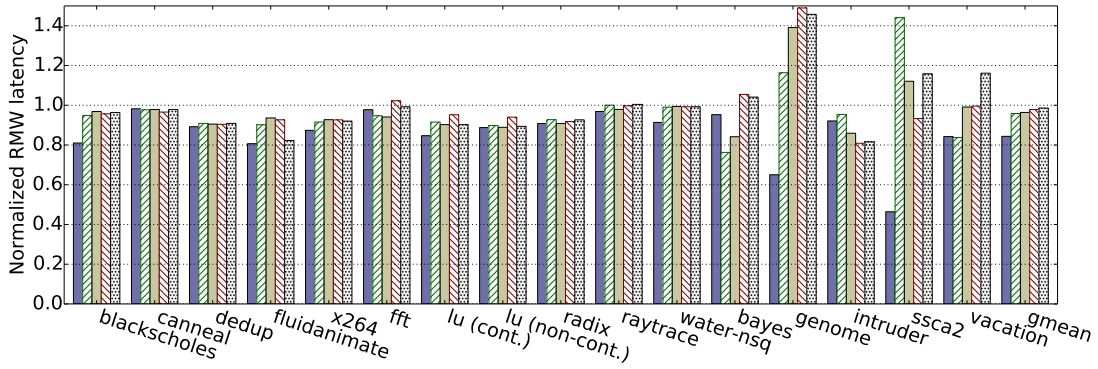


Figure 4.9: RMW latencies, normalized against the MESI baseline protocol. (Legend same as Figure 4.4, TSO-CC only)

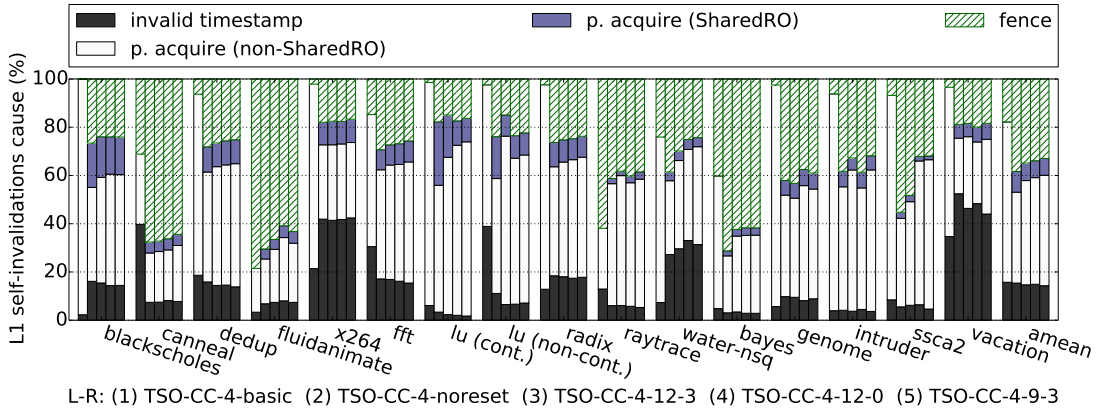


Figure 4.10: Breakdown of L1 self-invalidation cause.

false-sharing *(cont.)* results in similar execution times.

Second, our protocol performs better for GetX requests (writes, RMWs) to shared cache lines, as we do not require invalidation messages to be sent to each sharer, which must also be acknowledged. This can be seen in the case of *radix*, which has a relatively high write miss rate as seen in Figure 4.6. Further evidence for this can be seen in Figure 4.9, which shows the normalized average latencies of RMWs.

As we have seen, the introduction of the transitive reduction optimization (§4.2.3) contributes a large improvement over TSO-CC-4-basic, and next we look at how varying the TSO-CC parameters can affect performance.

TSO-CC-4-12-0: Decreasing the write-group size by a factor of 8× (compared to TSO-CC-4-12-3) results in a proportional increase in timestamp-resets, yet potential acquires detected are similar to TSO-CC-4-12-3 (Figure 4.8). One reason for this is that a write-group size of 1 results in more accurate detection of potential acquires, reducing self-invalidations. Thus, average execution time is similar to TSO-CC-4-12-3. However,

network traffic is more sensitive; TSO-CC-4-12-0 requires 5% more network traffic compared to the baseline.

TSO-CC-4-9-3: Decreasing the maximum timestamp size by 3 bits while keeping the write-group size the same, compared to TSO-CC-4-12-3, results in an expected increase of timestamp-resets of 8×, and stays the same compared to TSO-CC-4-12-0. Because of this, and because write-groups are more coarse grained, this parameter selection results in an increase of self-invalidations by 5% (7%), yet no slowdown compared to TSO-CC-4-12-3 (TSO-CC-4-12-0). The best case is *intruder* with an improvement of 24% over the MESI baseline, the worst case is *canneal* with a slowdown of 15%. TSO-CC-4-9-3 requires 7% more network traffic compared to the MESI baseline, indicating that network traffic is indeed more sensitive to increased self-invalidations.

As both timestamp-bits and write-group size change, the number of timestamp-resets in the system change proportionally. As timestamp-resets increase, invalidation of entries in timestamp-tables increases, and as a result, upon reading a cache line where there does not exist an entry in the timestamp-table for the line's last writer, a potential acquire is forced and all Shared lines are invalidated. This trend can be observed in Figure 4.8. The breakdown of self-invalidation causes can be seen in Figure 4.10.

4.5.1 Discussion

As highlighted via the results thus far, we have seen that TSO-CC performs well with traditional synchronization as used in PARSEC and SPLASH-2 (consistent results across workloads), as well as with less traditional synchronization such as software transactional memory used in STAMP (albeit with less consistent results across workloads).

In the following we will briefly discuss potential pathologies. First, and foremost, we note that the propagation delay for Shared data in TSO-CC is less predictable and higher compared to the baseline. This explains the less consistent results across STAMP benchmarks, but would also impact other workloads that benefit from best-effort propagation of modifications such as lock-free, racy codes. In particular asynchronous iterative algorithms [Bau78; VKG14], where the algorithm convergences faster if updated values are propagated faster may not perform as well as with an eager protocol.

On the other hand, server workloads as characterised by Ferdman et al. [Fer+12] would see little change compared to the baseline protocol. In particular, Ferdman et al. [Fer+12] note "on-chip and off-chip bandwidth requirements of scale-out workloads

are low" due to little sharing and communication but high read-only working sets, in particular instruction data. Since TSO-CC will classify read-only data as such, and will not be subject to self-invalidation, there would be no noticeable change compared to the baseline protocol: indeed, the same benefits (no self-invalidation of read-only data) but also pathologies (SharedRO is inclusive) will manifest.

4.6 Related Work

Closely related work is mentioned in previous sections, whereas this section provides a broader overview of more scalable approaches to coherence that precede development of TSO-CC.

4.6.1 Coherence for Sequential Consistency

Several approaches optimize the data structures and directory organization to maintain the list of sharers in eager protocols more efficiently; the protocols retain compatibility with sequential consistency.

Optimizing sharing vectors, hierarchical directory organizations [MHS12; Wal92] solve some of the storage concerns, but unfortunately increase overall organization complexity through additional levels of indirection.

Coarse sharing vectors [GWM90; ZSD10] reduce the sharing vector size, however, with increasing number of cores, using such approaches for all data becomes prohibitive due to the negative effect of unnecessary invalidation and acknowledgement messages on performance. More recently, SCD [SK12] solves many of the storage concerns of full sharing vectors by using variable-size sharing vector representations, but again with increased directory organization complexity.

Cuckoo directory [Fer+11] and SCD [SK12] optimize standalone sparse directory utilization by reducing set conflict issues. This allows for smaller directories even as the number of cores increase. Note that these approaches are orthogonal to our approach, as they optimize directory organization but not the protocol, and thus do not consider the consistency model.

Rather than maintaining the list of sharers in a bit vector at a central directory, the Scalable Coherent Interface (SCI) [GL96; Jam+90] maintains the list of sharers in a *distributed directory*—more specifically as a doubly-linked list, where each cache line tag also stores the next and previous processor and the memory points to the head

processor. SCI requires traversing this linked list sequentially to invalidate all sharers, which has a large worst-case overhead with a large number of sharers compared to TSO-CC which relies on self-invalidation. Furthermore, with an optimized directory (e.g. Cuckoo directory [Fer+11]), storage overheads for TSO-CC would be less than SCI due to SCI's use of both a next and previous pointer per private cache line.

Works *eliminating sharing vectors* [Cue+11; Pug+10], observe most cache lines to be private, for which maintaining coherence is unnecessary. For example, shared data can be mapped onto shared and private data onto local caches [Pug+10], eliminating sharer tracking. However, it is possible to degrade performance for infrequently written but frequently read lines, suggested by our implementation of CC-shared-to-L2.

4.6.2 Coherence for Relaxed Consistency Models

Dubois and Scheurich [DSB86; SD87] first gave insight into reducing coherence overhead in relaxed consistency models, particularly that the requirement of "coherence on synchronization points" is sufficient. Instead of enforcing coherence at every write (also referred as the SWMR property, see §3.2), recent works [ADC11; Cho+11; FC08; KK10; Liu+12; RK12; SKA13] enforce coherence at synchronization boundaries by self-invalidating shared data in private caches.

Dynamic Self-Invalidation (DSI) [LW95] proposes self-invalidating cache lines obtained as tear-off copies, instead of waiting for invalidation from directory to reduce coherence traffic. The best heuristic for self-invalidation triggers are synchronization boundaries. Based on [LW95], Lai and Falsafi [LF00] improve detection of synchronization points through trace correlation. More recently, SARC [KK10] improves upon these concepts by predicting writers to limit accesses to the directory. Both [KK10; LW95] improve performance by reducing coherence requests, but still rely on an eager protocol for cache lines not sent to sharers as tear-off copies.

Several recent proposals *eliminate sharing vector* overheads by targeting relaxed consistency models; they do not, however, consider consistency models stricter than RC. DeNovo [Cho+11], and more recently DeNovoND [SKA13], argue that more disciplined programming models must be used to achieve less complex and more scalable hardware. DeNovo proposes a coherence protocol for data-race-free (DRF) programs, however, requires explicit programmer information about which regions in memory need to be self-invalidated at synchronization points. The work by [RK12], while not requiring explicit programmer information about which data is shared nor a directory

with a sharing vector, present a protocol limiting the number of self-invalidations by distinguishing between private and shared data using the TLB.

Several works [MB92; YMG96] also make use of timestamps to limit invalidations by detecting the validity of cache lines based on timestamps, but require software support. Contrary to these schemes, and how we use timestamps to detect ordering, the hardware-only approaches proposed by [NN94; Sin+13] use globally synchronized timestamps to enforce ordering based on predicted lifetimes of cache lines.

4.6.3 Distributed Shared Memory (DSM)

The observation of only enforcing coherent memory in logical time [Lam78] (causally), allows for further optimizations. This is akin to the relationship between coherence and consistency given in §3.4. Causal Memory [AHJ91; Aha+95] as well as [KCZ92] make use of this observation in coherence protocols for DSM. Lazy Release Consistency [KCZ92] uses vector clocks to establish a partial order between memory operations to only enforce completion of operations which happened-before acquires.

4.7 Conclusion

We have presented TSO-CC, a lazy approach to coherence for TSO. Our goal was to design a more scalable protocol, especially in terms of on-chip storage requirements, compared to conventional MESI directory protocols. Our approach is based on the observation that using eager coherence protocols in the context of systems with more relaxed consistency models is unnecessary, and the coherence protocol can be optimized for the target consistency model. This brings with it a new set of challenges, and in the words of Sorin et al. [SHW11] “incurs considerable intellectual and verification complexity, bringing to mind the Greek myth about Pandora’s box.”

The complexity of the resulting coherence protocol obviously depends on the consistency model. While we aimed at designing a protocol that is simpler than the MESI baseline, to achieve good performance for TSO, we had to sacrifice simplicity. Indeed, TSO-CC requires approximately as many combined stable and transient states as the MESI baseline implementation.

Aside from that, we have constructed a more scalable coherence protocol for TSO, which is able to run unmodified legacy codes. Various verification efforts (including McVerSi proposed in Chapter 6) give us a high level of confidence in its correctness.

More importantly, TSO-CC has a significant reduction in coherence storage overhead, as well as an overall reduction in execution time. Despite some of the complexity issues, we believe these are positive results, which encourages a second look at consistency-directed coherence design for TSO-like architectures. In addition to this, it would be very interesting to see if the insights from our work can be used in conjunction with other conventional approaches for achieving scalability.

The next chapter introduces RC3, an extension of TSO-CC for a variant of RC on x86-64. RC3 demonstrates how the coherence protocol can exploit the explicit synchronization information present in modern language-level consistency models such as C11/C++11 and Java to achieves further storage savings while retaining comparable performance; like TSO-CC, RC3 also retains backward compatibility with TSO codes.

Chapter 5

RC3: Consistency Directed Cache Coherence for x86-64 with RC Extensions

5.1 Introduction

In recent years we have seen widespread convergence towards clearly defined programming language level memory consistency models, such as C11 [ISO11a], C++11 [ISO11b; BA08] and Java [MPA05]. These programmer-centric models require the programmer to explicitly distinguish and label data and synchronization operations at a much higher level of abstraction, rather than having to deal with the low level details of the hardware level consistency models (see §2.5). It is beneficial to convey the language level labels to the hardware, as hardware can exploit this information for improved performance. Since data operations need not be ordered among themselves, there are fewer restrictions on, e.g. out-of-order pipeline implementations.

In addition to the performance benefits, cache coherence protocol implementations in multiprocessor systems can also exploit synchronization information, leading to more scalable protocols. Indeed, with synchronization operations exposed, coherence need only be enforced *lazily* at synchronization boundaries via self-invalidation (see §3.4). Using self-invalidation, instead of relying on *eager* invalidations, is beneficial, as it no longer requires maintaining a sharing vector and associated data structures for maintaining the list of sharers. Although there have been numerous approaches to optimize the cache and directory organization to maintain the list of sharers more efficiently [Cue+11; Fer+11; GWM90; MHS12; SK12; Wal92], for coherence proto-

cols that exploit synchronization information, the sharing vector can be completely eliminated.

5.1.1 Motivation

Conveying data/synchronization information from the language level to the hardware level, however, requires a compatible hardware memory consistency model that also clearly distinguishes between data and synchronization operations. One such model, enabling an efficient mapping from the language to the hardware level, is RC (§2.4.3). In fact, a number of recent lazy coherence protocols [ADC11; Cho+11; FC08; KK10; RK12; SKA13] target variants of RC.

Unfortunately, some existing architectures such as x86 only support stricter memory consistency models, e.g. x86-TSO [OSS09], which cannot directly exploit the explicit data/synchronization information available at the language level. As there exists a well established ecosystem of software around these architectures, moving to a weaker RC variant is not an option as legacy code must continue to work. Therefore, most lazy coherence protocols cannot be applied to these architectures. One exception to this is the proposed TSO-CC protocol (Chapter 4), which implements a lazy coherence protocol for Total Store Order (TSO). Although TSO-CC enables lazy coherence for x86 systems, there is still no way to exploit the synchronization information available at the language level.

Therefore, our research question is the following: how can architectures (such as x86) benefit from the explicit synchronization information available from language level memory consistency models? At the same time, legacy code which assumes the original hardware memory model (x86-TSO), must continue to work. We attack this problem for the widely deployed x86-64 architecture.

5.1.2 Approach

In x86-TSO, reads and writes already provide acquire and release semantics respectively. Therefore, instead of adding additional acquire/release instructions to the ISA, we propose adding *ordinary* (relaxed) reads and writes to represent data operations (§5.3). This is realized via unused (null) prefixes which have become available in x86-64; the semantics of one unused prefix is changed to denote ordinary memory operations. The reads and writes from older legacy codes (that are not labeled with the extension) simply cause fewer instruction reorderings as the reads and writes are treated as acquires and

releases, just as is the case in x86-TSO. The resulting memory consistency model is *RCtso* (x86-RCtso). While variants of RC, such as RCsc and RCpc can be found in the literature [Gha+90], RCtso is not explicitly mentioned. RCtso is similar to RCpc in that it relaxes the release-acquire ordering, but unlike RCpc, preserves multi-copy atomicity for releases.

To take advantage of RCtso, we propose the *RC3* coherence protocol (§5.4): a lazy cache coherence protocol that targets RCtso. We base RC3 on TSO-CC, as it provides an efficient lazy coherence protocol implementation for TSO. In RC3, however, we additionally exploit the exposed ordinary/synchronization information to optimize the protocol. In TSO, since synchronization information is unavailable, every read can potentially be an acquire. TSO-CC employs transitive reduction using timestamps to limit self-invalidation: upon an L1 miss, self-invalidation is only performed if the response's timestamp is larger than the last-seen timestamp of the writer. There is, however, a significant cost to performing this optimization: to achieve good performance, each cache line in both L1s and L2, needs to hold a timestamp due to the absence of explicit synchronization information. In RC3, with data and synchronization information directly available, we no longer need to self-invalidate on ordinary reads. We observe, however, that there are performance gains to be realized when applying a limited form of transitive reduction optimization only to synchronization accesses, thereby reducing self-invalidations on redundant acquires, compared to a conventional RC lazy coherence protocol. Since synchronization accesses are relatively infrequent, we can perform this limited form of transitive reduction with *only per-L1 timestamps*, eliminating per cache line timestamps in both L1s and L2.

5.2 Limitations of TSO-CC

TSO-CC's first insight is that it is legal for a read to return a stale (locally cached) value. Periodic reads to a location eventually return the up-to-date copy of the value; TSO-CC accomplishes this by forcing a miss after a fixed number of hits—this effectively ensures the write propagation requirement of TSO. TSO's ordering requirements are not violated even though stale accesses are permitted by treating read misses (i.e. upon returning an up-to-date value) as acquires, which are followed by self-invalidation.

Considering every such read miss to be an acquire, however, causes excessive self-invalidations and degrades performance. *Which reads should be treated as acquires?* The second important insight concerns how to reduce excessive self-invalidations in the

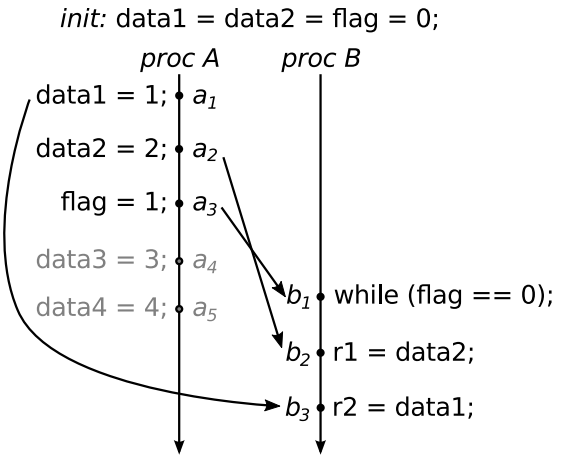


Figure 5.1: Extended producer-consumer example.

absence of explicit synchronization. To avoid redundant self-invalidations, TSO-CC proposes to use *transitive reduction* of acquires. Every L1 maintains a monotonically increasing timestamp source, which are then associated with writes. This information can then be used to answer the question "is there potentially stale data in my cache?", and decide if self-invalidation of shared data is required. TSO-CC needs to *apply transitive reduction at cache line granularity* (writer timestamp per cache line), because TSO-CC cannot distinguish between synchronization and non-synchronization. If synchronization and non-synchronization data writes (that map to distinct cache lines) would share the same timestamp, *timestamp false sharing* would limit the effectiveness of the transitive reduction optimization.

Using the example in Figure 5.1, TSO-CC ensures TSO as follows. Upon writes, increasing timestamps are assigned to the cache lines of data1, data2 and flag. Reading flag at event b_1 in processor B hits up to the maximum threshold, after which a miss is forced; this miss ensures that the most up-to-date value of flag is eventually read, and also causes self-invalidation of all other shared lines (in particular those containing data1 and data2). This miss also observes flag's timestamp, and processor B now associates this latest timestamp with processor A. Subsequent reads to data1 and data2 miss and obtain the (correct) up-to-date values, thereby ensuring the read-read order—if self-invalidation had not taken place, processor B would have observed stale copies of data1 and data2, violating read-read ordering. Due to transitive reduction, the misses at b_2 and b_3 do not cause self-invalidation, as the timestamps of the received data1 and data2 are both less than the timestamp of the already observed flag. Therefore, after the final event b_3, all of data1, data2 and flag are cached in B. To illustrate why maintaining timestamps at fine granularity is necessary, assume that processor A writes to several

other locations following a_3, and these additional writes' timestamps are shared with data1 and data2, but B has not yet observed flag. In this case, the timestamp associated with the write of flag may be lower than that of data1 or data2, and these misses would in fact cause self-invalidation—this is avoided by using timestamps at cache line granularity. With RC3, we show that per cache line timestamps are unnecessary if synchronization operations are explicitly exposed.

5.3 x86-RCtso: Release Consistency for x86-64

With the extra data/synchronization information available at the language level, how to expose this information to existing architectures with stricter consistency models? We propose a solution for the widely deployed x86-64 architecture, via extending the memory model from TSO to *RCtso*, a memory model which differentiates data and synchronization. In extending the memory consistency model of an architecture, a major objective is to retain backward compatibility with existing legacy codes as well as legacy microarchitectures (run new code on old systems). Specifically, in TSO [OSS09] reads and writes already provide acquire and release semantics respectively. Therefore, reads and writes from legacy TSO codes must retain their original semantics.

Accordingly, we ensure that existing reads and writes retain acquire and release semantics, but add support for the missing relaxed *ordinary* memory reads and writes. The resulting memory consistency model is RCtso, which is similar to RCpc [Gha+90] in that it relaxes the release-acquire ordering, but unlike RCpc, requires *write/multi-copy atomicity* of read-acquires and write-releases, which it inherits from the original TSO model. Table 5.1 provides an informal overview of the ordering constraints enforced by RCtso. The following is a formal definition in the framework described in §2.2.

Definition 5.1 (RCtso)**.** Let AS be the relation linking all read-acquires a to a write releases s (see Definition 2.9 for definitions of AM and MS): $\mathsf{AS} \triangleq \mathsf{AM} \cap \mathsf{MS}$. The relation mfence captures release-acquire pairs separated by a fence instruction. RCtso is instantiated as follows:

$$\begin{aligned} \mathsf{ppo} &\triangleq ((\mathsf{po} \cap \mathsf{AM}) \cup (\mathsf{po} \cap \mathsf{MS})) \setminus \mathsf{AS} \\ \mathsf{fences} &\triangleq \mathsf{mfence} \\ \mathsf{prop} &\triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{fr} \end{aligned}$$

Note that our variant of RCtso does not distinguish between special *sync* (acquire, release) and *nsync* operations [Gha+90]. Therefore, for ensuring correctness the compiler

Table 5.1: RCtso ordering requirements.

happens-before ↱	Read-Acquire	Write-Release	Read-Ordinary	Write-Ordinary
Read-Acquire	X	X	X	X
Write-Release		X		
Read-Ordinary		X		
Write-Ordinary		X		

will have to treat nsync reads and nsync writes conservatively as sync acquires and sync releases respectively. This primarily concerns *racy programs*: these continue to work in RCtso, as long as racy accesses are marked as synchronization. In programmer-centric models, such as C11/C++11, such accesses require special annotation (e.g. C11/C++11 atomics) irrespective of the hardware level model, and correctness of such codes is not affected as long as the compiler provides a conservative mapping to acquires/releases.

5.3.1 ISA Extension Details

This section describes the details of an ISA extension for the x86-64 architecture, effectively changing the supported memory consistency model from x86-TSO to x86-RCtso. In order to add the proposed relaxed ordinary memory operations, we have to label them explicitly. We can do so using instruction prefixes for memory operations. In x86-64 a group of prefixes, which were previously used for 32-bit mode to denote segment register overrides (CS, DS, ES, SS), have become unused and their meaning was changed to null prefixes [Adv13, §B.7].

Any one of these prefixes can be reused and their semantics changed from null to denote relaxed *ordinary* memory operations. In doing so, the ISA would not break compatibility with existing legacy codes, as unprefixed loads and stores retain their acquire and release semantics; these programs would merely impose a stricter program ordering among instructions. This also means that existing legacy synchronization libraries are compatible with new codes that make use of the extension to RCtso. Finally, this approach also ensures that new codes targeting RCtso are compatible with legacy microarchitectures, as in this case the prefix would revert to a null prefix. Therefore, the imposed program ordering will only be stricter than required, ensuring correctness [AG96].

5.4 RC3: Protocol Design

Our primary technical contribution is the RC3 protocol which takes advantage of the explicit labeling. This section describes the detailed protocol design: first we give an overview of the protocol (§5.4.1); this is followed by a detailed description of the basic RC3 protocol without optimizations (§5.4.2), and continue extending the basic protocol with the transitive reduction (§5.4.3) and shared read-only optimizations (§5.4.5). Throughout, the organization chosen assumes private L1 caches per core, and a tiled (NUCA) shared L2 with an embedded directory (§5.4.8).

5.4.1 Overview

We base the protocol on TSO-CC, as outlined in §5.2, and modify the protocol to exploit the fact that RCtso conveys synchronization and ordinary operations to the hardware explicitly. By exploiting this additional information, our goal is to further reduce the storage overheads of the resulting RC3 protocol, but retain comparable performance characteristics.

As the protocol is already aware of acquires and releases, we add support for the new ordinary memory operations. Upon acquires, where the last writer is not the requester, the protocol self-invalidates all shared cache lines in the local cache. It is worth noting, however, that self-invalidation is not required upon ordinary reads. Furthermore, shared lines fetched by ordinary memory operations can hit indefinitely in the local caches. In addition, we retain the TSO-CC optimization which permits acquires to hit shared cache lines up to a maximum number of accesses, as this ensures adequate performance for legacy codes.

In order to achieve good performance, TSO-CC proposes the transitive reduction optimization at cache line granularity. This is necessary, as newer writes (to different cache lines) after a write-release will be assigned increasing timestamps, but each write retaining a distinct timestamp value (until another write to the same line) due to using timestamps at cache line granularity. As timestamps assigned to older write-releases on different cache lines are unaffected until another release, unnecessary self-invalidations are rare. TSO-CC is effectively sharing timestamps at cache line granularity; at this granularity *timestamp false sharing* can only happen for all addresses mapped to a single cache line.

With RCtso, however, the protocol is explicitly conveyed information about synchronization and data accesses, and because data accesses dominate, self-invalidation

is suppressed for these accesses regardless. In the earlier example (§5.2) illustrated with Figure 5.1, using RCtso allows the read in B of flag to be marked as an acquire, and data1/data2 marked as ordinary reads. In this case, self-invalidation can only take place at b_1, but not b_2 or b_3 even if the timestamps of data1/data2 were higher than of flag—assuming shared timestamps and continued writes after the release of flag in A. Therefore, maintaining timestamps at cache line granularity is overkill, as explicit synchronization is infrequent and limited to relatively few addresses for which timestamps can be shared. Our hypothesis is, that applying timestamps at entire address-space granularity with RCtso optimized workloads is sufficient to realize the same performance benefits of transitive reduction as TSO-CC (validated in §5.6). Consequently, we can eliminate per cache line timestamps from L1s and L2 tiles, and only require maintaining per-L1 timestamps. In particular, per-L1 timestamps are still effective at reducing redundant acquires, e.g. due to conservative synchronization and acquiring mostly shared read-only data.

Furthermore, the protocol requires changes to the shared read-only optimization, as per cache line timestamps were previously used to *decay* lines from shared-written back to shared read-only. Our approach here is to reuse data structures already present in TSO-CC, but used for timestamp resets; specifically, we reuse the epoch-id, and only maintain epoch-ids per L2 cache lines to identify that a period of time has elapsed since the last write. As the epoch-ids require substantially less space then timestamps, this optimization, given its performance benefits, can be justified.

5.4.2 Basic Protocol

The following outlines the stable states, actions and transitions of the protocol.

Stable states: The protocol distinguishes between invalid, private and shared states. Cache lines in the L1 can be in invalid (Invalid), private (Exclusive, Modified, Exclusive_L, Modified_L) and shared (Shared, Shared_L) states. In the L2, private (Exclusive) cache lines only require a pointer b.owner to the current owner; shared (Shared) cache lines are untracked in the L2, and do not require tracking a list of sharers. The L2 maintains an additional state Uncached for cache lines not present in any L1, but valid in the L2.

We must introduce pairs of states in the L1: the base state, and a state (∗_L) denoting the line was fetched due to a reLaxed ordinary memory operation. This distinction is required to deal with cases where an ordinary memory operation caused a miss, but

followed by a synchronization hit. In the following we refer to the set of states with a common label prefix as Prefix∗, e.g. the set of states Exclusive and Exclusive_L are referred to as Exclusive∗. A transition from Exclusive∗ to Modified∗ means the transition is to the state with the same suffix (if any).

Read-Ordinary: Read requests (GetS) to cache lines invalid in the L2 cause an Exclusive response to the requesting L1, which must then acknowledge the response and transitions to Exclusive_L. If the cache line is in state Exclusive in the L2, the GetS request is forwarded to the current owner. The owner will then downgrade its copy from Exclusive∗ or Modified∗ to Shared∗. The owner responds to the initial requester with the data, which transitions to Shared_L; the owner additionally sends acknowledgement (if Exclusive∗) or data (if Modified∗) to the L2, which transitions the cache line to the Shared state. On subsequent read requests to the L2, the L2 responds with Shared data. Ordinary read accesses to Exclusive∗, Modified∗, and Shared∗ cache lines *always hit* in the L1.

Read-Acquire: Similarly to an ordinary read operation, a GetS request is sent to the L2. Upon receipt of a response, the L1 transitions to the respective base state, Exclusive or Shared.

As shared lines are untracked in the L2, all shared lines in the L1 must eventually be self-invalidated. To maintain the acquire-read and acquire-acquire orderings, L1s *self-invalidate all Shared∗ cache lines after every L1 synchronization miss, where the transition is to a base state, and the response's last writer is not the requesting L1*.

Read-acquire accesses hit to private lines (Exclusive_L, Modified_L) fetched due to an ordinary memory accesses, but are forced to perform self-invalidation of shared cache lines, as ordinary reads do not cause self-invalidation. This is, as outlined above, to address the corner case where an ordinary memory operation fetched a cache line, but the same cache line is subsequently accessed by a synchronization operation. After self-invalidation, the cache line is transitioned to the base state (e.g. from Exclusive_L to Exclusive). A read-acquire accessing a cache line in Shared_L causes a miss, as the cache line is most likely stale.

Read-acquire accesses to Shared cache lines are allowed to hit, but only up to a predefined maximum number of accesses, at which point a miss is forced. This requires extra storage for the access counter b.acnt. We reuse this optimization from TSO-CC, as firstly it provides adequate performance for legacy codes optimized for TSO. Secondly, this is the reason why the release-acquire ordering is relaxed in RC3, and thus targets RCtso.

Write-Ordinary: An ordinary write operation can only hit in the L1 if the line is held in the Exclusive∗ or Modified∗ states. Transitions from Exclusive∗ to Modified∗ are silent. An ordinary write misses in the L1 in any other state, causing a GetX request sent to the L2. Upon receipt of a response, the local cache line's state changes to Modified_L, the data is written to the L1, and an acknowledgement is sent to the L2. The L2 cache updates the cache line's state to Exclusive and updates b.owner with the requester's id.

If another L1 requests write access to a private line, the L2 forwards the request to the owner stored in b.owner, which then invalidates the line and passes ownership to the requester. Since the L2 only responds to write requests if it is in a stable state, i.e. it has received the acknowledgement of the last writer, there can only be one writer at a time. This serializes all writes to the same address at the L2 cache.

Upon a write request to a Shared line, the L2 immediately responds with a data response message and transitions the line to Exclusive. Note that even if the cache line is in Shared, the L2 must send the entire line, as the requesting core may have a stale copy. On receiving the data message, the L1 transitions to Modified_L either from Invalid or Shared∗. Note that there may still be other copies of the line in Shared∗ states in other L1 caches, but since they will eventually miss due to self-invalidation, and also cause self-invalidation of shared lines on synchronization misses, RCtso is satisfied.

Write-Release: Write releases hit in the same states as ordinary writes. Given release-acquire is relaxed, hits in the Exclusive_L or Modified_L states do not cause self-invalidation, and are treated as in the ordinary write case. Upon a write release miss, the final state upon receipt of a response is Modified; as per the rules outline above, such a miss would also cause self-invalidation.

Evictions: Inclusivity must be maintained for cache lines which are tracked by the L2: on evictions from the L2, evictions from Exclusive (and later SharedRO, see §5.4.5) require invalidation requests to the owner. Shared lines are untracked, and therefore evicted silently from the L2. Evictions from the L1 in states Exclusive∗ and Modified∗ require updating the L2 accordingly, which then transitions the line to Uncached; Shared∗ lines are evicted silently.

5.4.3 Opt. 1: Reducing Self-Invalidations of Redundant Acquires

In order to satisfy the acquire-read ordering, the basic protocol applies self-invalidation of Shared∗ lines at L1 misses. However, subsequent acquires would always cause

self-invalidation. If a release has already been observed, and all memory operations before it have previously been made visible via self-invalidation, self-invalidating again—upon acquiring the same, or any release that happened before it—is not required. To reduce unnecessary invalidations, we apply a variant of transitive reduction [Net93] like TSO-CC, but limited to synchronization misses.

Each L1 maintains a local *current timestamp* cur_ts of fixed size. The size of the timestamp depends on the storage requirements, but also affects the frequency of the timestamp resets, which is discussed in more detail in §5.4.4. The L1 local timestamp must be *incremented on every release*.

Upon propagating a cache line to the L2 cache, the L1's current timestamp cur_ts is propagated. The L2 then updates its respective entry for the sender in a last-seen timestamp table ts_L1. Note that, if we have multiple L2 tiles, the protocol requires a timestamp table per L2 tile. Each L1 also maintains a last-seen timestamp table ts_L1. The maximum possible entries per timestamp table can be less than the total number of cores, but will require an eviction policy to deal with limited capacity. The L2 responds to requests with the data, the writer b.owner and the last writer's most recent timestamp ts_L1[b.owner].

Thus, to reduce invalidations, only where the *L2's last-seen timestamp is larger than the L1's last-seen timestamp of the writer of the requested line*, treat the event as a *true acquire* and self-invalidate all Shared$*$ lines.

For those data responses where the timestamp is invalid (never written to since the L2 obtained a copy) or there does not exist an entry in the L1's timestamp-table (never read from the writer before), a self-invalidation is necessary. This is because timestamps are not propagated to main-memory and it may be possible for the line to have been modified and then evicted from the L2.

In case of an ordinary access miss followed by a read-acquire hit to the same line, transitive reduction cannot be directly applied (since the second access being a hit does not involve a response with a timestamp). However, we can still apply the transitive reduction as follows: on an ordinary access response, we check for *true acquire*, and if the check *would have caused self-invalidation*, we proceed to transition to the relaxed state, otherwise to the corresponding base state. This may still cause unnecessary self-invalidations where a synchronization miss (timestamp larger than last seen, causes self-invalidation) to a different line happens between the ordinary miss and the acquire hit (timestamp would have been less than or equal to last seen). Fortunately, this case is infrequent according to our evaluation.

5.4.4 Timestamp Resets

Because timestamps are finite, we have to deal with timestamp resets. Given the maximum timestamp size is chosen appropriately, and as they are only incremented on releases, resets should occur infrequently. If the current timestamp cur_ts is exhausted, L1s must broadcast a timestamp reset message to all L1s and L2 tiles. Upon receiving a timestamp reset message, an L1 invalidates the sender's entry in the timestamp table ts_L1; similarly for each L2 tile.

Handling races: It is possible for timestamp reset messages to race with data request and response messages: the case where a data response with a timestamp from a previous epoch arrives at an L1 which already received a timestamp reset message needs to be accounted for. The protocol requires maintaining *epoch-ids* per L1. The epoch-id of an L1 is incremented on every timestamp reset and the new epoch-id is sent along with the timestamp reset message. It is not a problem if the epoch-id overflows, as the only requirement for the epoch-id is to be distinct from its previous value. However, we assume a bound on the time it takes for a message to be propagated, and it is not possible for the epoch-id to overflow and reach the same epoch-id value of a message in transit.

Each L1 and L2 tile maintains a table of epoch-ids for every other L1. Every data message that contains a timestamp, must now also contain the epoch-id of the source of the timestamp. Upon receipt of a data message, the L1 compares the expected epoch-id with the data message's epoch-id: if they do not match, the same action as on a timestamp reset has to be performed, and can proceed as usual if they match.

Epoch optimization: As the current epoch is known to L2 tiles via the epoch-id table they maintain, we can make use of the epoch-id information to convey a more precise ordering than simply responding with the last-seen timestamp. This optimization requires addition of a small amount of extra storage for the written epoch-id to each L2 cache line.

If we know that the last writer's current epoch-id is different from the L2 cache line's epoch-id, the write must have happened before the last timestamp reset. In this case, the cache line's window for assigning the last-seen timestamp has *expired*. Upon cache line expiry, it is sufficient to assign the smallest valid timestamp to the response, so that we can avoid self-invalidation where the release has happened before the most recent release—under the assumption that the requesting L1 has already seen a more recent timestamp from the last writer.

One additional case must be dealt with: if the smallest valid timestamp is used in case of cache line expiry, it should not possible for an L1 to skip self-invalidation due to the line's timestamp being equal to the smallest valid timestamp. To address this case, the next timestamp assigned to a request response after a reset must always be larger than the smallest valid timestamp.

5.4.5 Opt. 2: Shared Read-Only with Epoch Based Decay

The basic protocol suffers from a pathological case, where shared cache lines which are written to very infrequently but read frequently are self-invalidated unnecessarily. TSO-CC greatly benefits from introducing the shared read-only optimization to avoid this, but makes use of per cache line timestamps in deciding when a shared line should be classified read-only. This section describes an alternative policy without full timestamps.

We add another state SharedRO for shared read-only cache lines, which are excluded from self-invalidation. A line transitions to SharedRO instead of Shared if the line is not modified by the previous Exclusive owner. Additionally, cache lines in the Shared state are transitioned (*decay*) to SharedRO upon *expiry*: *if the cache line's written epoch does not equal the last writer's current epoch* (see §5.4.4). The L1s maintain SharedRO and SharedRO_L states, where the request was either due to synchronization or an ordinary operation respectively. On an acquire to a SharedRO_L line, the L1 must self-invalidate shared lines, followed by the line transitioning to SharedRO—as described above, the prior ordinary access does not cause self-invalidation.

In the case of a synchronization access to a SharedRO cache line where the last writer is not known, the L1 would always have to perform self-invalidation. Similar to TSO-CC, we can introduce L2 SharedRO timestamps, where each L2 maintains a current timestamp. As we do not store timestamps in cache lines, a SharedRO response is assigned the L2's current timestamp. On a cache line transitioning from Exclusive or Shared to SharedRO, the L2 tile increments its current timestamp. Each L1 must maintain a table ts_L2 of last-seen timestamps for each L2 tile. On receiving a SharedRO response from the L2, the following rule determines if self-invalidation must occur: if the line's timestamp is *larger than the last-seen timestamp from the L2*, self-invalidate all Shared$*$ lines. Furthermore, to reduce the number of L2 timestamp increments, the L2's current timestamp is not incremented if there does not exist a cache line which transitioned (since the last increment) to a state from which SharedRO can be reached (for the specific rules, see §4.2.4).

Upon resetting an L2 tile's timestamp, a broadcast is sent to every L1, and the L1s remove the entry in ts_L2 for the sending tile. As outlined in §5.4.4, to avoid races, L2s also maintain epoch-ids, and every L1 maintains a table of epoch-ids epoch_ids_L2. To avoid sending larger timestamps than the current timestamp, we again apply the epoch optimization.

Writes to SharedRO: A write request to a SharedRO line triggers broadcast invalidate, and subsequent acknowledgements. Network traffic can be reduced by reusing the b.owner bits as a broadcast filter as described in §4.2.4. SharedRO evictions from L1 are therefore silent, but evictions from L2 requires broadcast invalidate, followed by acknowledgements.

5.4.6 Atomic Instructions and Fences

Implementing atomic read and write instructions, such as RMWs, is trivial with the proposed protocol: each atomic instruction issues a GetX request. Fences require unconditional self-invalidation of cache lines in the Shared state. Note that in our implementation, fences do not invalidate cache lines fetched by ordinary memory operations (Shared_L), which implies that fences do not enforce ordering between ordinary memory operations (as per Definition 5.1).

5.4.7 Speculative Execution

In the presence of a speculative execution pipeline, the same rules as outlined in §4.2.7 apply. Furthermore, RCtso not only permits optimizations in the coherence protocol, but also closer to the core pipeline. Ordinary writes do not require being retired from the store-buffer in strict FIFO order. Furthermore, a load-buffer may elide squashes (and deem speculation correct) for invalidations of cache lines due to ordinary reads. Gharachorloo, Gupta, and Hennessy [GGH91] provide more detail on optimizations for RC.

5.4.8 Storage Requirements and Organization

Table 5.2 shows a detailed breakdown of storage requirements for RC3, referring to literals that have introduced throughout §5.4. We assume a local L1 cache per core and a NUCA [KBK02] architecture for the shared L2 cache.

Like TSO-CC, even though a sparse directory embedded in the L2 cache was chosen

Table 5.2: RC3 specific storage requirements.

L1	**Per node:** • Current timestamp cur_ts, B_{ts} bits • Current epoch-id cur_eid, $B_{epoch-id}$ bits • Timestamp-table ts_L1$[n]$, $n \leq C_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries Only required if SharedRO opt. (§5.4.5) is used: • Timestamp-table ts_L2$[n]$, $n \leq C_{L2-tiles}$ entries • Epoch-ids epoch_ids_L2$[n]$, $n = C_{L2-tiles}$ entries **Per line b:** • Number of accesses b.acnt, B_{maxacc} bits
L2	**Per tile:** • Last-seen timestamp-table ts_L1$[n]$, $n = C_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries Only required if SharedRO opt. (§5.4.5) is used: • Current timestamp, B_{ts} bits • Current epoch-id, $B_{epoch-id}$ bits • Increment-timestamp-flags, 2 bits **Per line b:** • Epoch-id b.epoch_id, $B_{epoch-id}$ bits • Owner (Exclusive), last writer (Shared), coarse vector (SharedRO) as b.owner, $\lceil log(C_{L1}) \rceil$ bits

for all evaluated configurations (§5.5.3), the protocol is independent of a particular directory organization and could be combined with more efficient organizations (see §4.2.8). The protocol does not require inclusivity for Shared$*$ lines, alleviating some of the set conflict issues associated with the chosen organization.

Table 5.3: System parameters.

Core-count & frequency	32 (out-of-order) @ 2GHz
Write buffer entries ROB entries	32, FIFO 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	1MB×32 tiles, 64B lines, 16-way 30 to 80 cycles
Memory Memory hit latency	2GB 120 to 230 cycles
On-chip network	2D Mesh, 4 rows, 16B flits
Kernel	Linux 2.6.32.61

By eliminating per cache line timestamps, we significantly simplify cache organization compared to TSO-CC. Notably, eliminating per cache line timestamps simplifies lookup of the timestamps as they no longer need to be associated with a particular address tag. Other structures such as the MSHR also no longer require a timestamp entry.

5.5 Evaluation Methodology

This section provides an overview of our evaluation methodology used in obtaining the performance results (§5.6). We also discuss storage overheads of the protocol configurations used in §5.5.3.

5.5.1 Simulation Environment

The environment is almost identical to the one used for the TSO-CC evaluation (§4.4), but with some subtle differences described in the following. Again, we use the Gem5 simulator [Bin+11] with Ruby and GARNET [Aga+09] in full-system mode. The ISA is x86-64 with RCtso extensions added (§5.3). The processor model used for each core is a simple out-of-order processor. Table 5.3 shows the key-parameters of the system. As the protocols evaluated explicitly allow accesses to stale data, we added support to the simulator to functionally reflect cache hits to stale data; unmodified, the used version of Gem5 in full-system mode would assume the caches to always be coherent

Table 5.4: Benchmarks and their input parameters

PARSEC	blackscholes	simmedium
	canneal	simsmall
	dedup	simsmall
	fluidanimate	simsmall
	x264	simsmall
SPLASH-2	fft	64K points
	lu	512×512 matrix, 16×16 blocks
	radix	256K, radix 1024
	raytrace	car
	water-nsquared	512 molecules
STAMP	bayes	-v32 -r1024 -n2 -p20 -i2 -e2
	genome	-g512 -s32 -n32768
	intruder	-a10 -l4 -n2048 -s1
	ssca2	-s13 -i1.0 -u1.0 -l3 -p3
	vacation	-n4 -q60 -u90 -r16384 -t4096

otherwise.

5.5.2 Workloads

Table 5.4 shows the benchmarks we have selected from the PARSEC [Bie+08], SPLASH-2 [Woo+95] and STAMP [Min+08] benchmark suites. The STAMP benchmark suite has been chosen to evaluate transactional synchronization compared to the more traditional approach from PARSEC and SPLASH-2; the STM algorithm used is NOrec [DSS10] as it is the current default.

In order to optimize the full-system software stack we use, we modified GCC's machine description for x86-64, which adds the chosen prefix (SS prefix) for all ordinary data operations. As all chosen workloads make clear use of synchronization libraries, we only had to make sure the synchronization libraries were unmodified, in effect using read-acquires and write-releases. We further optimized as many system libraries of the distribution as possible.

All selected workloads correctly run to completion with the evaluated protocol configurations. The program codes are unmodified, but targeting x86-64 with RC extensions (§5.3). The Linux kernel used, however, is unmodified and compiled *without* RC extensions, as we ran into limitations of our ad-hoc conversion from TSO to RCtso. This means that our results are conservative, and a system with a fully optimized

software stack will yield the same or better performance as our evaluation shows. A rigorous conversion of x86-TSO optimized codes to x86-RCtso is beyond the scope of this thesis. With our conversion, the total size of all workload binaries increases by 7%.

5.5.3 Protocol Configurations and Storage Overheads

As before, we compare against the existing Gem5 implementation of the MESI baseline directory protocol (§3.3.1). The configuration we use for TSO-CC is the same as the one determined optimal in the limited design space exploration in §4.4. We include the shared read-only optimization as described in §5.4.5 in all non-MESI configurations. Note that, we do not include a version of RC3 with infinite timestamps, as this renders the SharedRO *decay* optimization ineffective due to non-resetting timestamps (epoch-id never changes). Our evaluation showed that a version of RC3 with infinite timestamps performs worse than or equal to a configuration of RC3 with finite timestamps—as such, we exclude this configuration. Below we consider the following configurations: RC-base, TSO-CC, RC3.

RC-base: A conventional RC protocol that removes the sharer list, and relies on self-invalidation of shared cache lines on acquires. Ordinary read misses do not cause self-invalidation. We derive RC-base's implementation from RC3, effectively a version without timestamps. This is to provide a fairer comparison, in particular so that RC-base's implementation includes the shared read-only optimization (however, lacking timestamps, without the ability to decay Shared lines). In this protocol, acquires always miss if the cache line is in Shared state. With the evaluated system configuration as seen in Table 5.3, RC-base reduces coherence storage requirements by 76% compared to the MESI baseline for 32 cores.

TSO-CC: This version is the overall best performing TSO-CC configuration as found in §4.5. This configuration uses 4 bits for the accesses counter, 12 bits for the timestamps and a 3 bit write-group counter. TSO-CC reduces storage requirements by 38% compared to the MESI baseline for 32 cores.

RC3: This is the RC3 protocol with all optimizations enabled. This configuration uses 4 bits for the accesses counter and 12 bit timestamps. Compared to the MESI baseline for 32 cores, this configuration of the RC3 protocol saves 66% on-chip storage, and 45% compared to TSO-CC. In addition to purely saving storage overheads, RC3 simplifies cache organization compared to TSO-CC, thereby saving power consumption; however, a detailed study of power consumption is beyond the scope of this thesis and reserved

Table 5.5: Coherence state storage overheads with all optimizations enabled: private L1 per core, 1MB per L2 tile, and as many tiles as cores; the timestamp-table sizes match the number of L1s and L2 tiles; $B_{epoch-id} = 3$ bits per epoch-id. Normalized w.r.t. the MESI baseline, coherence storage MB.

Cores	32	64	128
MESI	100% (2.13)	100% (8.27)	100% (32.53)
TSO-CC	62% (1.33)	34% (2.80)	18% (5.91)
RC3	34% (0.73)	19% (1.59)	11% (3.49)
RC-base	24% (0.52)	14% (1.16)	8% (2.56)

for future work. We include **RC3-legacy** to show the performance of *legacy codes* with the RC3 protocol. In this configuration, the ISA extension is not used for the workloads.

Table 5.5 shows a comparison of the extra coherence storage requirements between the MESI baseline, TSO-CC, RC3 and RC-base (in order of decreasing storage requirements). With the chosen configurations, RC3 reduces on-chip storage requirements by 89% (41%) over the MESI baseline (TSO-CC) for 128 cores.

5.6 Experimental Results

The goal of our evaluation is to analyze the performance characteristics of RC3 in comparison with the MESI baseline protocol, a conventional RC baseline and TSO-CC. Our initial hypotheses are as follows. Firstly, we expect that RC3, with the help of the transitive reduction optimization (albeit with per-L1 timestamps), will perform significantly better than RC-base. Secondly, despite using only per-L1 timestamps, we expect RC3 to perform as well as TSO-CC (which uses per cache line timestamps), as it can additionally leverage explicit synchronization information. In the following, we will validate our hypotheses by comparing the performance and network overhead of RC3 with that of RC-base and TSO-CC (and the MESI baseline). In order to isolate the contribution of synchronization information in RC3, we will also compare against RC3-legacy, which is identical to RC3 in all respects, except that it is not conveyed explicit synchronization information. The analysis focuses on performance results in Figure 5.2 showing execution times, and Figure 5.3 showing network traffic; we use supporting data from Figures 5.4 and 5.5 which show cache hit/miss rates, and Figure 5.6 showing total self-invalidations.

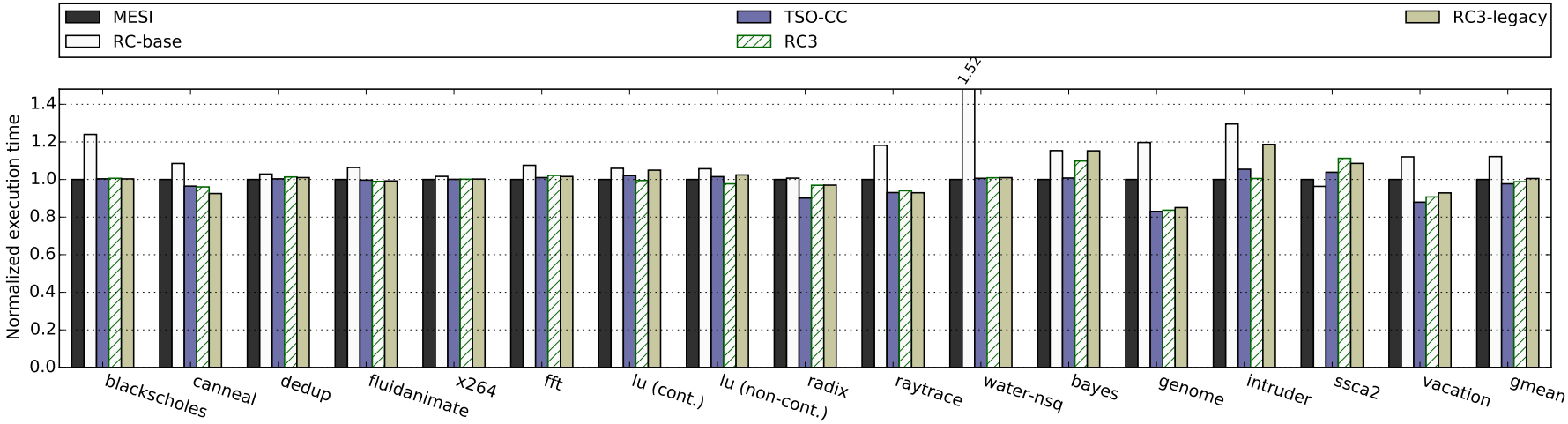


Figure 5.2: Execution times, normalized against the MESI baseline protocol.

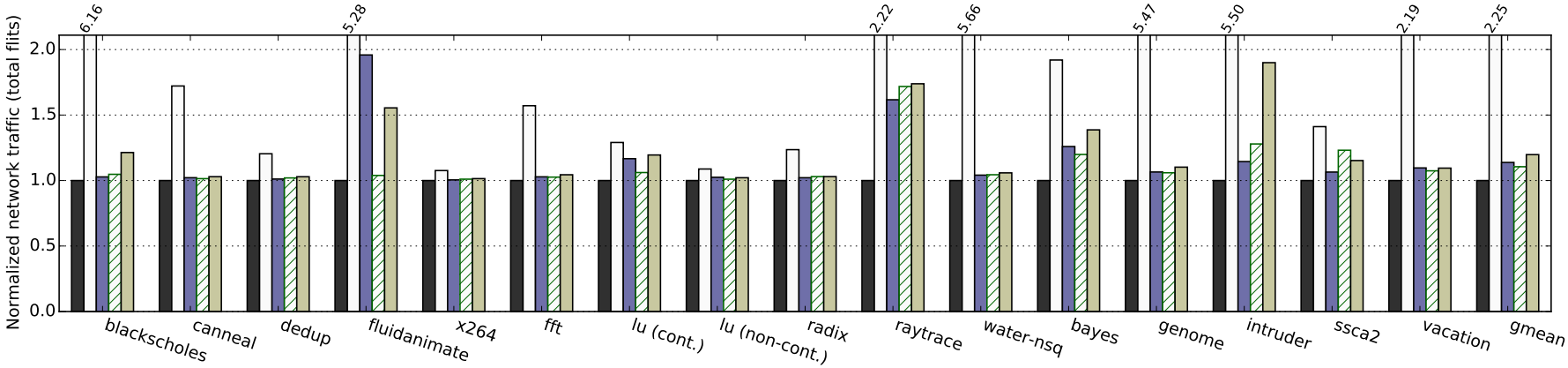


Figure 5.3: Network traffic (total flits), normalized against the MESI baseline protocol.

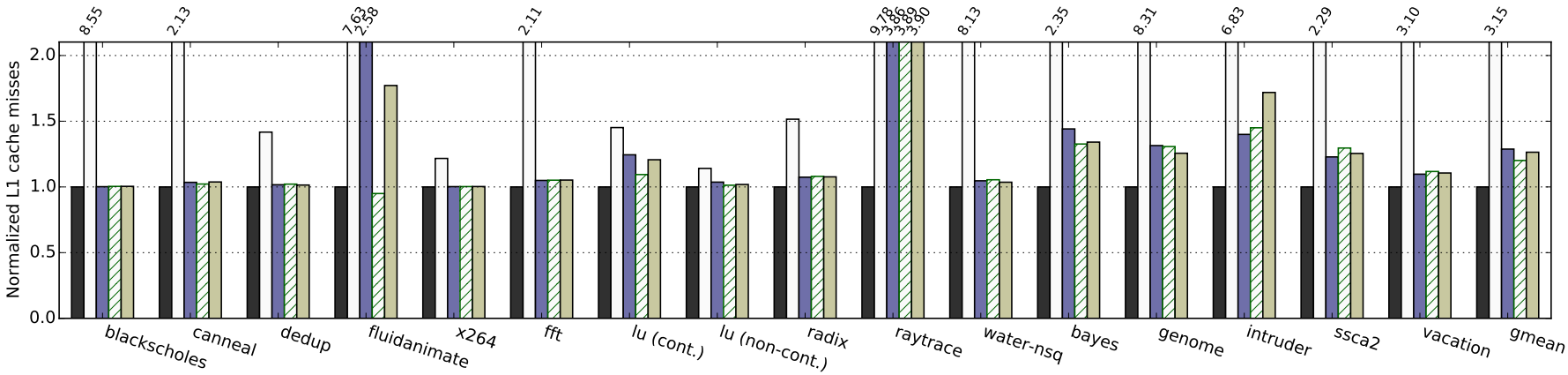


Figure 5.4: L1 cache misses, normalized against the MESI baseline protocol.

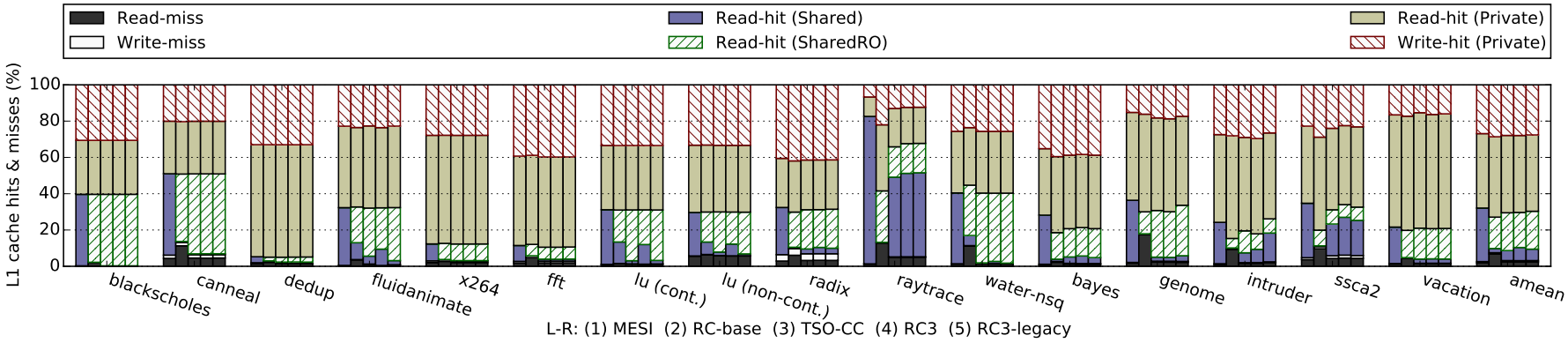


Figure 5.5: L1 cache hits and misses; hits split up by Shared, SharedRO and private (Exclusive, Modified) states.

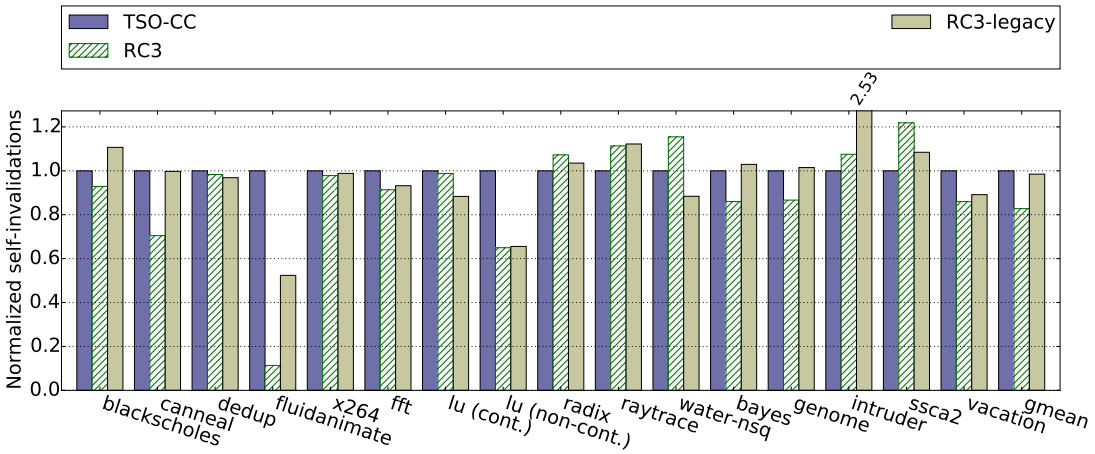


Figure 5.6: Normalized self-invalidations w.r.t. TSO-CC.

With explicit synchronization information, is transitive reduction using timestamps still required for performance? In order to answer this question, we compare the performance of RC-base with that of RC3. As seen in Figure 5.2, on average the baseline RC protocol RC-base causes a slowdown of 12% compared to the MESI baseline protocol. Network traffic (Figure 5.3) is far more sensitive, with an average increase of 125% compared to the MESI baseline. Interestingly, the network traffic as well as L1 misses (Figure 5.4) are heavily correlated, yet often with much less noticeable effects on execution times, as the out-of-order cores can hide miss latencies well. Introducing the optimizations of RC3 provides an average improvement over RC-base of 12% in terms of execution times, and 57% in terms of network traffic. RC3 reduces redundant acquires via the transitive reduction optimization, and most of the difference can be attributed to the consequent reduction of self-invalidations: compared to RC-base we note a reduction of self-invalidations by 800% on average. However, why does RC3 perform poorly in the first place with respect to self-invalidations? We believe this is due to redundant acquires in RC-base, an artifact of overly conservative synchronization in parallel codes [RG01]. RC3 solves this problem via transitive reduction. This validates our first hypothesis, that RC3 outperforms RC-base, and therefore transitive reduction improves performance even where explicit synchronization information is provided to the protocol.

We note that write misses do not vary much across configurations, and the biggest difference in performance is due to read misses. Firstly, this is due to the fraction of reads (avg. 70%) dominating that of writes (avg. 30%), and secondly because writes are not in the critical path as they are entered into a write buffer. Furthermore, write misses

due to downgrades are infrequent because of relatively small number of communicating accesses, in particular for SPLASH-2 and PARSEC benchmarks [BFM09].

With explicit synchronization information, how does removing per cache line timestamps (and instead only rely on per-L1 timestamps) affect performance? In order to answer this question, we compare the performance of RC3 with that of TSO-CC (and the MESI baseline protocol). RC3 performs as well, in terms of execution times, as TSO-CC and the MESI baseline on average. The best case execution time is achieved with *genome*, which improves by 16% over the MESI baseline; in the worst case we observe a slowdown of up to 11% for *ssca2*. Figure 5.6 shows total self-invalidations normalized against TSO-CC: on average, RC3 self-invalidates 17% fewer cache lines compared to TSO-CC. By reducing self-invalidations, RC3 reduces L1 misses (Figure 5.4) by 7% compared to TSO-CC; thereby RC3 reduces network traffic by 3% over TSO-CC. This validates our second hypothesis that RC3 performs at least as well as TSO-CC and the MESI baseline; even though RC3 only uses per-L1 timestamps, it leverages explicit synchronization information to self-invalidate fewer cache-lines.

Without explicit synchronization information, how does removing per cache line timestamps (and instead only rely on per-L1 timestamps) affect performance? To answer this, we compare the performance of RC3-legacy with that of TSO-CC and RC3. On average, execution times of RC3-legacy are very close to TSO-CC and RC3, but network traffic increased by 5% and 8% respectively. Indeed, self-invalidations (Figure 5.6) appear to be on-par with TSO-CC, but 20% higher than RC3. However, we see higher variance across benchmarks. In particular for some STAMP benchmarks, RC3 is significantly better—in the case of *intruder*, RC3-legacy increases execution time by 17% and network traffic by 48%.

From this study we can observe that, for workloads with relatively frequent synchronization such as *intruder* and *bayes* in STAMP, more precisely identifying synchronization either via exposing synchronization (RC3) or using fine grained timestamps (TSO-CC) is important. However, for other benchmarks (e.g. most from PARSEC and SPLASH-2), where time spent communicating is relatively low, even with per-L1 timestamps but no explicit synchronization information (RC3-legacy), performance is good. In these cases, the protocol is efficient at properly classifying (see Figure 5.5) private and shared read-only data which are excluded from self-invalidation.

5.7 Related Work

Some of the broader related work have already been mentioned in §4.6, but is again summarized with a focus on the context of RC3.

5.7.1 Language to Hardware Level Consistency

RCtso has not been explicitly mentioned in the literature, although variants of the RC memory model have been formally defined in the literature [Gha95]. In particular, the RCpc consistency model, also relaxes the release to acquire ordering; in contrast to RCtso, however, RCpc but does not require multi-copy atomicity among releases and acquires. While not explicitly referred to as RCtso, Intel Itanium implements what we consider RCtso [Int02].

The definitions of Adve et al.'s *data-race-free* [Adv93; AH90; AH93] and Gharachorloo et al.'s *properly labeled* [Gha+90; Gha95] form the basis for the programmer-centric models we use in our discussion to highlight the fact that the programmer does not need to be exposed to the complexity of the resulting hardware level consistency model. Our work takes a more practical approach, proposing a detailed implementation of the memory consistency model in an existing architecture, and how the previously stricter (x86-TSO) consistency model can be extended (x86-RCtso).

Some existing architectures have started to provide support for achieving a mapping from a language level model to the hardware memory model, that lets it retain synchronization information. For example, the ARMv8 architecture [ARM14] has introduced releases and acquires into the ISA. In contrast with ARM, where the resulting extended model (via adding releases and acquires) is stronger than the original model, the case for x86 is more challenging as the extended model RCtso is weaker than the original model; legacy issues arise in the latter but not the former.

Note also that recent Intel processors have introduced hardware transaction extensions (including XACQUIRE, XRELEASE) [Int14]. However, these are for a different purpose, namely lock elision [RG01]. Our problem is orthogonal, as we are interested in weakening the memory consistency model; in this instance, we argue that since TSO reads and writes already have acquire and release semantics respectively, exposing relaxed memory operations is the right approach. It is worth noting that in the same way as hardware transaction extensions were introduced in a backward compatible way, we propose reuse of a null prefix on memory operations to introduce more relaxed ordinary memory operations.

5.7.2 Consistency Directed Coherence

Several recent works [ADC11; Cho+11; FC08; KK10; RK12; SKA13] target relaxed memory consistency models, typically RC or Weak Ordering [AG96]; DeNovo [Cho+11] and DeNovoND [SKA13] follow a programmer-centric approach (SC for DRF). These works introduce a number of optimizations for enhancing the performance of relaxed consistency protocols. Notably, optimizing higher-level synchronization primitives (locks, barriers, etc.) [Cho+11; SKA13; RK15] would help improve latencies and reduce misses, as polling behavior could be avoided. These optimizations, however, are orthogonal to our proposal, as we stuck with implementations of current operating system and standard library vendors. Unfortunately, none of these approaches can directly be applied to existing architectures with stricter models.

SPEL [RJ15] is a dual-consistency protocol, which can guarantee SC, and provide performance improvements given explicit code annotations denoting DRF. Although legacy compatible, the protocol does not reduce storage overheads.

Coherence for GPUs has become a recent topic of interest, to more efficiently support wider ranges of workloads. GPUs are typically programmed using higher level languages (e.g. OpenCL), and the vendor is responsible for a correct mapping to the hardware level. As such, the system-centric memory consistency model of GPUs has not been readily exposed. However, recent proposals for coherent memory systems on GPUs propose RC [Sin+13; Hec+14].

5.7.3 Data Structures in Eager Protocols

Numerous works attack the cache coherence problem by optimizing the data structures and cache & directory organization to maintain coherence state—in particular the list of sharers more efficiently via: hierarchical directory organizations [MHS12; Wal92]; sharing vector compression [GWM90; Zeb+09]; variable size sharing vectors [SK12]; or optimizing directory utilization [Fer+11; SK12] (for more detail, see §4.6.1).

While most of these approaches are not directly applicable in protocols without a list of sharers, some can also be applied to different protocols (such as the proposed RC3)—in particular, those that optimize directory utilization (e.g. Cuckoo [Fer+11]). None of these approaches consider the memory consistency model explicitly. Unlike these approaches, we propose changing the protocol, and by optimizing for the memory consistency model, to only require less costly data structures in the first place. By combining directory optimization approaches and the RC3 protocol, the potential on-

chip storage savings can be even greater.

5.8 Conclusion

In recent years we have seen widespread convergence towards clearly defined programming language level memory consistency models. Each of these models requires the programmer to explicitly distinguish between synchronization and data memory operations. If such a language level model is mapped to a compatible lower level hardware consistency model that preserves the synchronization information, the additional information can then be exploited by hardware for enhanced performance and scalability. In particular, we have seen a resurgence on the study of lazy cache coherence protocols that exploit this explicit labeling of synchronization and data to achieve scalable coherence protocols. Most of these proposals assume (variants of) RC, which inherently differentiates between data and synchronizations.

There are however existing architectures, which support hardware consistency models that do not directly allow for synchronization information to be conveyed. To make matters worse, some of these architectures (most notably x86) support stricter models. It is not possible for such architectures to transition to RC overnight, as legacy code written assuming the stricter model should continue to work. This chapter has presented a viable way to achieve this transition for x86-64, by addressing:

(1) how synchronization information from the language level can be exposed to the hardware;

(2) how cache coherence can take advantage of this information.

We have shown synchronization information can be conveyed relatively easily (and elegantly) by simply conveying whether or not a memory operation is a relaxed operation using unused prefixes in the ISA. We have then shown how the cache coherence protocol can be designed to take advantage of the relaxations, yet ensure TSO for legacy codes. All this with significant storage savings in comparison to not only the MESI baseline but also TSO-CC, a lazy coherence protocol designed to target TSO. Performance of RC3 is significantly better than baseline RC as we eliminate redundant acquires. Despite using only per-L1 timestamps (as opposed to per cache line timestamps employed by TSO-CC), RC3's performance is comparable to TSO-CC (and the MESI baseline) as we exploit synchronization information.

The following chapter proposes McVerSi, a new approach for rigorous simulation-based memory consistency verification of a full-system. Its need was realized from the verification challenges of consistency-directed protocols (TSO-CC, the basis for RC3, is used as a case study) and the interaction with other components; note however, that even conventional protocols and their interaction with the rest of the system is an open problem, highlighted by the discovery of new bugs in Gem5's MESI protocol.

Part IV

Memory Consistency Verification

Chapter 6

McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation

6.1 Introduction[1]

The relationship between weaker MCMs and processor implementations can be seen as a "chicken-and-egg" problem. While many existing weak MCMs are the product of desired microarchitectural optimizations (MCM formalized after implementation), it is equally desirable that new optimizations do not violate a specified MCM of an *architecture*. For example, write-buffers, in the absence of any other visible optimizations, give rise to TSO, as in e.g. x86 (see §2.4.2). On the other hand, this thesis proposes designing *consistency directed* cache coherence protocols, e.g. TSO-CC (Chapter 4) has been designed specifically with TSO in mind.

Problems arise, however, when the programmer believes they are working with a particular model, but the hardware exhibits behavior weaker than the promised MCM: either the model is incorrect, or the hardware contains bugs—both scenarios are undesirable. Recent work has uncovered problems with deployed CPUs [AMT14], and GPUs [Alg+15] using litmus testing. While a proof of MCM correctness of the *functional design implementation* (e.g. cycle accurate model) would provide the highest

[1] For brevity and clarity, this chapter abbreviates "memory consistency model" as MCM throughout.

possible guarantees, unfortunately, the complexity to achieve such a feat is usually not cost-effective [Hie+09]. Nonetheless, there exists a wealth of literature on memory system and MCM verification, which all help to raise the designer's confidence in the implementation.

Various methodologies can be applied at different stages of the design development. In the *pre-silicon* design phase, approaches based on formal methods are usually applied to *abstract models* of *components* of a design. For example, model checking of coherence protocols has been studied extensively [ASL03; CSG02; Che+06; CMP04; Jos+03; KAC14; McM01; PD97]; consistency properties verified are commonly based on derived properties, such as the Single-Writer–Multiple-Reader (SWMR) invariant (see §3.2). Another example is the recently developed PipeCheck [LPM14] tool, which is a domain-specific MCM model checker for pipeline abstractions. Applying any of these techniques to as many components as possible is essential, to avoid an implementation based on faulty component specifications.

In the final design implementation, however, the composition and interaction of the components must remain safe with respect to the MCM. With individual component verification, this is often overlooked. For example, one of the bugs discovered in our evaluation (MESI,LQ+IS,Inv[1]) would not have been discovered through individual verification of either pipeline or coherence protocol.

Consequently, *full-system MCM verification* is required. This has arguably been achieved in the *post-silicon* environment using a testing based approach [Alg+12; Che+09; DWB09; Han+04; MH05; Roy+06]. Tests consist of threads of instruction sequences, which are executed in the full-system and their results checked for adherence to the MCM. There are various approaches to test generation, ranging from random [Han+04] to user-directed [Alg+12]. Post-silicon approaches can afford to execute large tests, as instruction throughput is much higher than in simulation. *Yet, while all these approaches could be made to work in a pre-silicon environment, i.e. full-system simulation, they would be too slow as the above approaches do not optimize test generation for simulation*.

6.1.1 Approach

Any verification approach strives to ensure an implementation's adherence to its high-level specification. Test based methods trade off a non-exhaustive (reduced states and

[1]The coherence protocol fails to forward invalidation to the Load Queue (LQ), leading to reordered reads (§6.5.3).

transitions covered) result for a more detailed implementation. Therefore, the goal of any test based method should be to *cover* as many states and transitions as possible, in order to provide the highest possible guarantees about the system in the absence of a proof [Hie+09; IE12]. To achieve this, we develop an approach to automatically improve test suitability for exposing MCM violations and guide tests towards unexplored states and transitions.

Our focus lies on automated *simulation-based verification of a full-system design implementation*: we propose McVerSi, a test generation framework for *fast, coverage directed MCM verification in simulation*. Using a Genetic Programming (GP) [Koz92] based approach, we show how to generate tests for a full-system simulation that achieve (§6.3):

(1) greater coverage of the system;

(2) and improved test quality specifically for MCM verification.

To achieve (1), we leverage the additional observability available in simulation (white-box) and use coverage as the GP *fitness function*. The designer of a system can select any number of suitable coverage metrics; in our implementation, we use the covered logic implementing the coherence protocol as the coverage metric (structural coverage), as the coherence protocol is crucial in enforcing the desired MCM, and is the source of some of the most elusive bugs. To achieve (2), we design a domain-specific GP *crossover function*. Our crossover is selective on memory operations contributing to high non-determinism of a test; highly deterministic tests are uninteresting from an MCM verification point of view, as few execution witnesses are invalid, and the probability of observing an invalid witness due to a bug is low.

The tests generated are compiled on-the-fly to the target ISA, which are then executed in the full-system. For a GP-based approach to quickly converge towards better tests, high test throughput is essential. Therefore, instead of large tests typically used with random tests (e.g. TSOtool [Han+04] shows results for $\geq$12k operations), we are interested in very short tests (e.g. in our evaluation we use 1k operations). Thus, the time from one test to the next must be minimized (overhead for checking, test generation, synchronization). To accelerate test execution, we introduce several extensions that any simulator to be used for verification must provide (§6.4). In particular, a host interface for the simulation-aware guest control program for configuring the test generator, emitting code on-the-fly, checking, and host-assisted barriers.

6.2 Evolutionary Algorithms

Evolutionary algorithms are heuristic search algorithms, a machine learning approach inspired by the principles of natural selection. Genetic Algorithms (GAs) [Hol75] are one such approach, with a broad range of optimizations problems where GAs have proven to be practical solutions [SP94]. The goal is to iteratively evolve an initially random population of *chromosomes* (or genomes) towards increasingly optimal solutions. In GAs, each chromosome is usually represented as a fixed-size string, which encodes values of the parameters to be searched. Each solution has a *fitness* value, determined via a domain-specific fitness-function. Based on the fitness values, *selection* then chooses several solutions to be used to generate new offspring. *Crossover* and *mutation* are the operators used to create offspring from the selected parents, with crossover choosing parts of each chromosome to be recombined into one or more children, and mutation selecting few individual genes to be modified.

Genetic Programming (GP) [Ban+98; Koz92] is an adaption of GAs, that instead of searching for strings of parameters, search for actual *executable programs* which yield *executable solutions* to the search problem. The general approach is like in GAs, but the representation and crossover of chromosomes is specialized to yield valid programs in the language and domain being targeted.

Machine learning approaches have been successfully applied to generate successively better tests to increase coverage across a wide range of microprocessor verification scenarios [IE12]. For example, GAs have been used to search for biases for pseudo-random test generators [Bos+01]. Using GP, μGP [CCS03] has been proposed to generate test programs directed by various coverage metrics. In this work we use a GP test generation approach, similar to μGP, but with a focus on multi-threaded test generation for the purpose of MCM verification.

6.3 Test Generation

This section describes the proposed automated test generation approach, whose goal is to reveal as many MCM bugs as fast as possible. Section §6.3.1 provides an overview; §6.3.2 discusses in more detail the mapping of coverage to fitness; and finally §6.3.3 describes test representation, crossover and mutation operators.

6.3.1 Overview

Given pseudo-randomly generated tests (instruction sequences), with some constraints given by the user (distribution of operations, memory address range, and stride), *how can the test generator improve tests without further user input?* Coverage, which refers to the fraction of system state explored, is a widely used metric to assess test quality [Hie+09; IE12; Bos+01; CCS03], giving an indication of how close the verification task is to completion. Over time, the test generator's goal should, therefore, be to direct tests towards rarely covered state transitions based on coverage feedback. In the absence of further information about the implementation, apart from coverage reports, the only input we will give the simulated system are instruction sequences. Finding a precise solution to cover rare state transitions given this degree of control is a complex problem, and approximate solutions based on *evolutionary algorithms* (see §6.2) have been used successfully.

McVerSi uses Genetic Programming (GP) [Koz92; Ban+98] based test generation. Tests (chromosomes) are *represented* as directed acyclic graphs (DAGs) of operations [CCS03]. Each node (gene) represents a high-level operation of a thread; each operation, in turn, maps to an executable representation in the target ISA. A *test-run* corresponds to executing the test for several iterations; after a test-run completes, the fitness of the run is evaluated and associated with the test.

As the goal of the test generator is to generate tests covering as many states and transitions of the system as possible, the *fitness function* is defined in terms of coverage. For the purpose of MCM verification, all crucial bits of logic affecting enforcement of the MCM should be captured by coverage. This, by and large, means the processor pipeline, coherence protocol and on-chip interconnect. In our implementation, we restrict coverage to structural (code) coverage of the coherence protocol, as the most challenging bugs we study are related to the coherence protocol; having said this, our framework is not tied to this choice and it is indeed possible to augment coverage with functional coverage metrics (e.g. store-buffer becoming full). Our coverage computation dynamically adapts such that frequent state transitions are excluded from coverage so that the focus shifts towards rare protocol transitions. In other words, the GP verification goals change over time.

However, in order to be able to detect MCM bugs in the first place, we require tests which are more likely to expose MCM violations; we will refer to this as *MCM test suitability*. This means we seek tests where a large fraction of possible candidate

init: x = 0, y = 0	
Thread 1	**Thread 2**
x ← 1	r1 ← y
y ← 1	r2 ← x

Figure 6.1: Message passing pattern.

executions are invalid under the specified MCM, to increase the probability of observing an invalid candidate execution due to a possible bug. To illustrate, Figure 6.1 shows the common message passing litmus test. Assuming e.g. a TSO model, the outcome $\mathsf{r1} = 1 \wedge \mathsf{r2} = 0$ is forbidden. If, however, we were to remove either write of x or y, all candidate executions become valid—such a test would not be very useful from an MCM verification perspective. While there are several ways to increase the probability of generating suitable MCM tests (e.g. constrain the usable address range), this would preclude us from generating tests which could expose bugs requiring large address ranges (e.g. due to cache evictions).

In order to be able to converge towards more suitable tests, first, we must be able to tell how suitable a given test is. Given that our tests far exceed the size of litmus tests, it would be too costly to enumerate all possible candidate executions of a test in order to determine the set of valid and invalid executions. Instead, we observe that tests with a large number of candidate executions are highly non-deterministic/racy. Therefore, to generate more suitable MCM tests, we should favor such tests. The key metric we introduce is the *average non-determinism of a test* ($\mathsf{ND_T}$), which informally is a measure of the average number of races observed per event (memory operation) across all iterations of the test-run. More precisely, it is the average number of events that are *conflict ordered before* any given event in the test.[2] A value of 1 means the test-run is not observed to be non-deterministic/racy, i.e. all events are only ordered after the initial events (e.g. "init" in Figure 6.1). A $\mathsf{ND_T}$ value larger than 1 implies that races have been observed.

Definition 6.1 (Conflict orders across runs)**.** Let i be the iteration in a test-run. The simulator records the conflict order relations rf_i and co_i (defined in §2.2) for each iteration. Then we define the *union of all iteration's observed conflict orders* to be

$$\mathsf{rfco_{RUN}} = \bigcup_i (\mathsf{rf}_i \cup \mathsf{co}_i)$$

[2] A prerequisite for this metric to be meaningful is that a test-run has more than one iteration.

Definition 6.2 (Average non-determinism of a test)**.** Let n be the *total events (memory operations) executed* in a test. We define the average non-determinism in a test as the cardinality of $\mathsf{rfco}_{\mathsf{RUN}}$ divided by n

$$\mathsf{ND}_{\mathsf{T}} = \frac{|\mathsf{rfco}_{\mathsf{RUN}}|}{n}$$

We initially assessed including ND_{T} in the fitness function, and using standard GP crossover operators [Koz92; Ban+98]. This, however, did not result in significantly more suitable tests over time. This is because, the non-deterministic result of a test is sensitive to the specific sequences and interaction of instructions: breaking them up without considering the key ingredients to the non-deterministic result caused little progress towards more suitable MCM tests. In other words, merely combining random instructions from two racy tests cannot guarantee a new more racy test—instead tests must be recombined in a way, such that the resulting test is likely to be more racy than its parents.

Instead, to generate more suitable MCM tests, we design a *selective crossover*, which gives preference to memory operations involved in races, i.e. those with observed non-deterministic results across several iterations. More specifically, our goal is to preserve sequences of memory operations on addresses (a key ingredient of MCM tests) which contribute highly towards non-determinism of the test outcomes. For this we measure the *non-determinism of each event (memory operation)* (ND_e), and give preference to those events whose non-determinism is higher than the test's ND_{T}.

Definition 6.3 (Observed event non-determinism)**.** We define the non-determinism of an individual event e_k as the cardinality of the set of events which are ordered before e_k (via conflict order) across a test-run

$$\mathsf{ND}_e = |\{e \mid \forall e \ : \ (e, e_k) \in \mathsf{rfco}_{\mathsf{RUN}}\}|$$

Further details of GP parameters, *selection* method and operations used, which are independent of our proposed scheme, are discussed in the evaluation (§6.5).

6.3.2 Coverage and Fitness

Coverage gives an indication of how close the verification task is to completion, and ideally lets us judge if all interesting scenarios that we think can lead to bugs have been covered. Therefore, we use coverage as the GP fitness function. Note, the verification

scenarios and therefore coverage goals are highly system dependent, and the following is one of many possible options.

In modern multiprocessors, the cache coherence protocol is a key component in the implementation of the MCM [SHW11]. The goal of the coherence protocol is to make caches transparent, such that data inconsistencies (and MCM violations) are not due to accesses to stale cached data. Because the hardest to find bugs we study originate in the coherence protocol, we use structural coverage over the protocol's possible state transitions as the *fitness function*.

More specifically, the simulator counts transitions, represented via the unique triple of origin state, action and next state; here the state is any possible state, including transient states. For our study, we do not distinguishing between identical controllers, and instead consider the sum of their transitions.

Each test fitness is assigned a coverage value independent of any prior run tests, i.e. only what has been covered by a particular test-run. Because the simulation is running continuously, loading new tests on-the-fly, any state changes of previous tests that affect following tests must be reset (e.g. flush caches—see §6.4) to produce consistent results.

Next, we do not always consider all protocol transitions, and instead frequent transitions are excluded from coverage, i.e. we compute an *adaptive coverage*. The goal of this is, since the simulation is running continuously, and thus recording all transitions since simulation start, we can direct the test population towards unexplored and rarer transitions. Effectively, this helps avoid getting a population stuck in a local maximum, where little progress is made towards unexplored states.

Upon initialization we consider those transitions, whose transition counts is less than a low initial cut-off value. If the adaptive coverage falls below a certain threshold for too many test evaluations, the cut-off is doubled (exponential increase). Then, if we consider a total of t transitions, and if in a test run n of these were covered, the fitness of that test would be n/t. Each test's fitness is evaluated only once.

6.3.3 Test Representation, Crossover and Mutation

Representation: Each test (chromosome) is represented as a DAG of a constant number of nodes (genes), which naturally represents control flow [CCS03] and each *disjoint sub-graph representing one thread*. A sequence of nodes corresponds to the program order of one thread. Each node is a high-level operation (`op`) of a thread which maps to executable code of the target ISA. Furthermore an `op` is responsible for the mapping

to one or more events in the MCM (only for memory operations) as per the defined instruction semantics.

Nodes are stored internally as a flat list of tuples: each tuple represents $\langle \mathsf{pid}, \mathsf{op} \rangle$, where pid is the processor/thread ID and op is an operation. The order of nodes within this list gives rise to the code sequence of instructions, but not necessarily program order (po), e.g. due to branches. The class and properties of an operation specifies how nodes are connected *upon* code generation, such that copying individual nodes of one thread to another (via crossover), forms another valid thread. The final DAG representation is restrictive enough to allow efficient generation of the static ordering relations of the target MCM, as well as an efficient crossover as described in the following.

Crossover: As outlined in §6.3.1, we determine that a standard crossover is unsuitable for the problem of generating more suitable MCM tests. We design a *selective crossover*, which selects and merges thread sub-graphs based on operations which highly contribute towards non-determinism of a test. The key metric we introduce to assess a test's degree of non-determinism is the *average non-determinism of a test* ($\mathsf{ND_T}$). After evaluation of a test-run, we obtain its $\mathsf{ND_T}$ (Definition 6.2) and each event's ND_e (Definition 6.3). From this, we obtain the set of events' addresses fitaddrs, where an event's ND_e *is larger than the rounded* ND_T *of the test*. The proposed *selective crossover* then always selects those nodes where the address of a memory operation is a member of the set of addresses fitaddrs.

To be able to place a bound on the simulated execution time of a test, we enforce the number of nodes of a test to be constant. Note, however, that the number of nodes per thread is not necessarily constant. Additionally, in MCM tests we would like to *preserve some of the relative scheduling properties* of test operations; e.g. an operation which is placed at the end of a thread should not be moved to the beginning of a thread in a new test after crossover. During recombination of two tests, the flat list representation of the DAG nodes simplifies enforcing both the above properties efficiently.

Mutation: Following crossover, *mutation* takes place if necessary, which mutates nodes by randomizing thread and operation, but preserving the relative position in the test. As not all operations are necessarily selected from either parent test, missing nodes are generated pseudo-randomly: this step already contributes to mutation, effectively enabling more directed mutation, such that in the early stages of test generation useful sequences of operations are retained.

Summary: Algorithm 1 shows our proposed crossover and mutation. Figure 6.2 illustrates test representation and their crossover.

Algorithm 1: Crossover and mutation.

Let P_{MUT} be the mutation probability;
Let P_{USEL} be the unconditional mem. op. selection probability;
Let P_{BFA} be the bias with which a new operation has an address from the set of `fitaddrs`;
Let `fitaddrs(`*test*`)` return the set of addresses of events, where $\mathsf{ND}_e >$ rounded ND_T of *test*;
Let `is_memop(`*op*`)` return **true** if *op* is a memory operation, **false** otherwise; where **true**, *op* has a valid attribute addr denoting the memory address accessed;
Let `random_bool(`*p*`)` generate a Bernoulli variate with probability *p*;

Function `fitaddr_fraction(test)` **begin** `/* Returns fraction of memory operations which are guaranteed to be selected. */`
  mem_ops ← [op|⟨pid, op⟩ ∈ test ∧ `is_memop(op)`];
  return $\frac{\texttt{len}([\mathsf{op}|\mathsf{op} \in \mathsf{mem\_ops} \wedge \mathsf{op.addr} \in \texttt{fitaddrs}(\mathsf{test})])}{\texttt{len}(\mathsf{mem\_ops})}$;

Function `crossover_mutate(test1, test2)` **begin**
  $a_1 \leftarrow$ `fitaddr_fraction(test1)`; $a_2 \leftarrow$ `fitaddr_fraction(test2)`;
  $P_{SELECT1} \leftarrow a_1 + P_{USEL} - (a_1 \cdot P_{USEL})$; $P_{SELECT2} \leftarrow a_2 + P_{USEL} - (a_2 \cdot P_{USEL})$;
  child ← test1; mutations ← 0;
  for i ← 0 **to** `len(child)` **do**
    ⟨pid, op⟩ ← test1[i];
    if `is_memop(op)` **then**
      select1 ← `random_bool(`P_{USEL}`)` ∨ op.addr ∈ `fitaddrs(test1)`;
    else
      select1 ← `random_bool(`$P_{SELECT1}$`)`;
    ⟨pid, op⟩ ← test2[i];
    if `is_memop(op)` **then**
      select2 ← `random_bool(`P_{USEL}`)` ∨ op.addr ∈ `fitaddrs(test2)`;
    else
      select2 ← `random_bool(`$P_{SELECT2}$`)`;
    if ¬select1 ∧ select2 **then**
      child[i] ← test2[i];
    else if ¬select1 ∧ ¬select2 **then**
      mutations ← mutations + 1;
      if `random_bool(`P_{BFA}`)` **then**
        child[i] ← Make random ⟨pid, op⟩, with addresses constrained to `fitaddrs(test1)` ∪ `fitaddrs(test2)`;
      else
        child[i] ← Make random ⟨pid, op⟩;
    else
      `/* Retain node` child[i]`. */`
  if mutations/`len(child)` < P_{MUT} **then**
    Mutate child with probability P_{MUT};
  return child

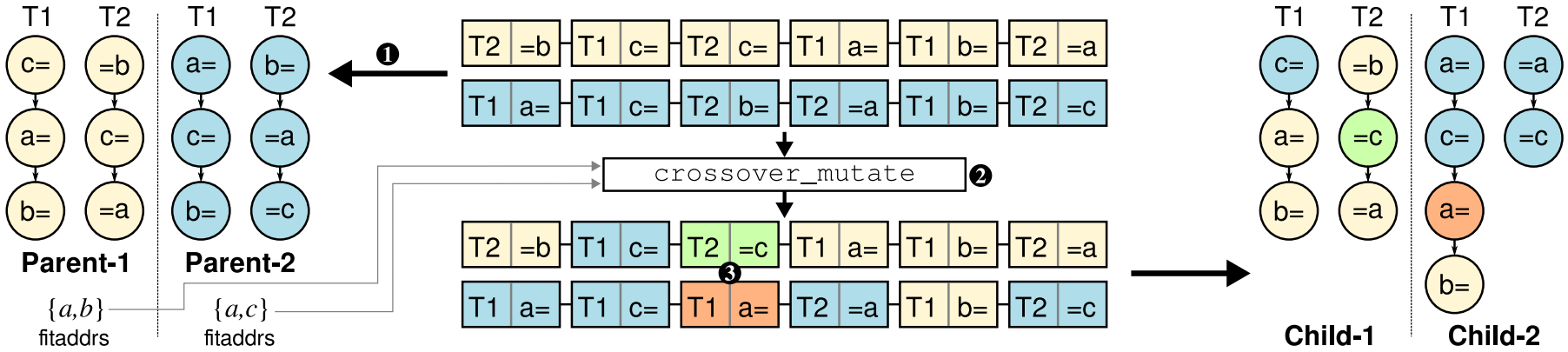


Figure 6.2: Crossover and mutation example. Initially two tests with two threads each, ❶ which are then evaluated and the set of fitaddrs determined to be $\{a,b\}$ for Parent-1 and $\{a,c\}$ for Parent-2. ❷ Given these two parents, crossover can produce several children, of which two are shown. ❸ Unselected addresses in the same slot for both parents result in mutation in this slot, and further mutation is no longer necessary.

6.4 Accelerating Test Execution & Checking

To allow a GP approach to progress towards more optimized tests as fast as possible, we must increase test throughput, i.e. minimize the wall-clock time for each test-run. As the tests are run in a full-system, a minimal guest workload is responsible for setup and running each test. We minimize the wall-clock time to execute code that is part of test setup and control, but does not contribute towards actual test execution. We propose several extensions, that any simulator to be used for verification should implement.

Table 6.1 shows the proposed interface between the simulation-aware guest workload and the host system.[3] Algorithm 2 shows the kernel of the guest workload, and is self-explanatory. While it is possible to implement many of the functions as part of the guest program (*optional* and *suggested* in Table 6.1), host-assistance transfers the implementation onto the simulation host system, thereby *speeding them up significantly*. In particular, we found that the host assisted barrier is a mandatory pre-requisite to execute very short tests, as the perturbation and thread offset induced by a guest barrier implementation was too large. With the host assisted barrier, thread start offset is minimized, and using very short tests becomes possible.

6.4.1 Checker

A pre-silicon environment, in this case simulation, provides certain advantages over post-silicon; most notably, we can afford to observe all necessary conflict orders to implement a polynomial-time decision procedure to verify if a recorded candidate execution object is valid or invalid with respect to the target axiomatic MCM [GK97]. Since axiomatic descriptions are effectively dealing graph based representations of executions, the core of our checker relies on a depth-first search (DFS) to query the graphs constructed.

The MCM descriptions that our checker accepts are based on the framework proposed by Alglave, Maranget, and Tautschnig [AMT14] and summarized in §2.2. The rationale behind this is that the precise and correct formalization of more complex MCMs (e.g. ARM or Power, or proposed GPU [Alg+15] models) should not be attempted in an ad-hoc manner, and by implementing the aforementioned framework we can afford a more direct mapping of these published MCMs to our checker. Note, however, that the style (axiomatic vs. operational) nor the particular formalization is a dependency for our proposed test generation scheme.

[3] Here we refer to *guest* as the system being simulated, and *host* the simulation software.

Table 6.1: Guest-Host interface.

Function	*Description*
barrier_wait_coarse()	*Host-assistance optional.* Barrier which does not mandate threads to be precisely synchronized.
barrier_wait_precise()	*Host-assistance suggested.* Barrier which mandates that threads are precisely synchronized via host-assistance or otherwise, such that upon return threads are in lock-step.
make_test_thread(*code*)	*Direct host-interface.* Host writes *code* for current test of thread.
mark_test_mem_range(*a*, *b*)	*Direct host-interface.* At guest workload initialization, use to set test generator address-range from start address *a* to end *b*.
reset_test_mem()	*Host-assistance suggested.* Resets (write initial values) locations used by test; flushes cache lines and other structures affecting following test executions.
verify_reset_all()	*Direct host-interface.* Verifies last test execution. Clear entire candidate execution object (static and conflict orders). *Evaluates test-run and sets up next test.*
verify_reset_conflict()	*Direct host-interface.* Verifies last test execution. Clears only conflict orders of candidate execution object.

All static orders required to compute the preserved program order (ppo) are gathered before first execution of a test. The DAG representation (§6.3.3) of a test makes this straightforward. Furthermore, before test execution, each write event is assigned a unique ID—the value to be written by the associated instruction—to be able to map observed values to a producing write. This implies that the size of each instruction can support the maximum desired writes; there is no limit for read count. Initially, all memory is zero, and upon reading the initial value, the initial write event is created on first use.

All dynamic orders (conflict orders rf and co) are observed during execution of a

Algorithm 2: Guest workload: per thread kernel. The *control thread* is a thread selected at program startup to drive the generate-verify-reset cycle.

Input: test_iterations denoting the execution count of a test per test-run.

```
/* Every thread has its own independent memory region,
   which is used by the host to copy the respective code
   for the thread.                                        */
code ← Allocate executable memory, host-writable;
while true do
    barrier_wait_coarse();
    make_test_thread(code);
    for i ← 0 to test_iterations do
        barrier_wait_precise();
        execute code;
        barrier_wait_coarse();
        if i + 1 < test_iterations ∧ is control thread then
            verify_reset_conflict();
            reset_test_mem();
    if is control thread then
        verify_reset_all();
        reset_test_mem();
```

test (without affecting functional execution). Constructing rf requires extracting the value an instruction reads; inserting into co requires extracting the value an instruction overwrites. In order to map *committed* instructions to an operation of a test, which then maps to an event in the MCM, we use the respective instruction pointers (IPs) to create a unique mapping. In case where an instruction can give rise to several reads and/or writes, we use the microcode counter to uniquely map to an event.

6.4.2 Complexity Implications

At the core of an axiomatic model checker (of an execution) is a graph-search algorithm, which is used to construct all required derived relations and then assert all constraints over these are satisfied. The complexity of checking a candidate execution against particular axiomatic models has been the primary concern of many post-silicon verifi-

cation works [Han+04; MH05; Roy+06]. Unlike post-silicon, however, a pre-silicon environment can afford a straight-forward, complete and polynomial-time decision procedure as all conflict orders are visible [GK97].

On the other hand, operational models are defined in terms of an abstract machine, which given a read, then specifies the set of possible values a read may observe. Their complexity advantages for simulation-based verification have been realized in past works [Sah+95; Sha+08], where such models are also referred to as *relaxed scoreboards*. Each transition is monitored by a checker, effectively ensuring that the simulated system only performs transitions which are legal according to the model.

In this work we describe MCMs in terms of axiomatic models (§6.4.1), as simulation together with short tests (§6.4) affords an efficient and relatively simple checker. Note, however, our main contribution concerns MCM test generation.

6.5 Evaluation Methodology

This section discusses the evaluation methodology used in obtaining the results (§6.6). The goal of the evaluation is to show the performance of the McVerSi framework with regard to its bug finding capability and the wall-clock time required to find a bug.

6.5.1 Simulation Environment

We evaluate our approach using the cycle accurate Gem5 simulator [Bin+11] with Ruby and GARNET [Aga+09] in *full-system* mode with the x86-64 ISA. It is worth noting that Gem5 is frequently used for pre-silicon design evaluation, with several industrial users. The processor model used for each core is a simple out-of-order processor. Table 6.2 shows the key-parameters of the system. All cache coherence protocol implementations are modeled in a functionally accurate manner (not just timing), to ensure that stale data (e.g. due to a protocol bug) affects functional execution.

To demonstrate that the bug finding ability of a particular test generator is consistent (statistically significant), we run each generator/bug pair 10 times with a time limit—this should provide high confidence in a test generator in case all runs find a bug found. To demonstrate that the convergence time of the GP-based approach is within practical bounds, each simulation run is limited to 24 hours of host time.[4] For non-GP test

[4]The host platform is server-grade, with Intel Xeon® E5620 CPUs; we measure 30k simulated instructions per second.

Table 6.2: System parameters.

Core-count & frequency	8 (out-of-order) @ 2GHz
LSQ entries ROB entries	32 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	128KB×8 tiles, 64B lines, 4-way 30 to 80 cycles
Memory Memory hit latency	512MB 120 to 230 cycles
On-chip network	2D Mesh, 2 rows, 16B flits
Kernel	Linux 2.6.32.61

generators, we shall note that this effectively translates to measuring the frequency of a bug found within 24×10 hours (10 days), as these test generators do not continuously update their internal state to progress towards better tests.

Each simulation run (out of 10) uses a different random seed for both simulation and test generation yielding different executions per sample. Furthermore, simulation startup overheads are negligible, as the simulation loads the guest workload (§6.4), and then runs it continuously until a bug is found or the time limit is reached. Upon reset after a test execution (one iteration of a test-run), non-test related simulation state is not reset; therefore, the following executions of the same test in the same simulation are all perturbed differently.

6.5.2 Test Generation & Checking

This section outlines the test generation and checking approaches we evaluate and compare.

6.5.2.1 McVerSi

To demonstrate the effectiveness of our proposed test generation approach, we compare against the following test generation variants. It is worth noting that each of the following still makes use of the simulation-specific optimizations (§6.4) of the McVerSi framework.

First, to show the effectiveness of pseudo-randomly generated tests, which most

Table 6.3: Test generation parameters.

Test size	1k operations (total across threads)
Iterations	10 test executions per test-run
Test memory (stride)	1KB (16B), 8KB (16B)
Operations:bias% (comment)	• Read:50% (read into reg.) • ReadAddrDp:5% (read into reg. with address dependency) • Write:42% (write from reg.) • ReadModifyWrite:1% (RMW, on x86 also implies fences) • CacheFlush:1% (cache flush, e.g. `clflush` on x86) • Delay:1% (constant delay using NOPs)
McVerSi-ALL, McVerSi-Std.XO	
Population size	100
Tournament size	2
Mutation probability (P_{MUT})	0.005
Crossover probability	1.0
McVerSi-ALL	
P_{USEL}	0.2
P_{BFA}	0.05

previous works (§6.7) rely on, we include the **McVerSi-RAND** configuration.

Next, we evaluate a naïve GP-based approach, **McVerSi-Std.XO**, which demonstrates the need for our domain-specific crossover. McVerSi-Std.XO does not make use of the selective crossover and instead, for all threads, connects sub-graphs, by removing a random vertex, of a thread from two parents; a standard single-point crossover over the flat list can be exploited to efficiently realize this. The fitness function is modified to include the additional objective for improving test suitability (equal weighting for coverage and normalized ND_T).

Finally, the configuration **McVerSi-ALL** includes all proposed test generation (§6.3)

and the simulation-specific optimizations (§6.4). Both McVerSi-ALL and McVerSi-Std.XO implement a *steady-state GA* with *tournament-selection* and the *delete oldest replacement* strategy [VF96]. It is worth noting that steady-state GAs have been shown to outperform generational GAs in dynamic or non-stationary environments [VF96].

To assess the effect of our proposed crossover function alone—i.e. to answer the question: *are highly non-deterministic tests alone sufficient?*—we did evaluate a configuration with a constant fitness function. While better performing than either McVerSi-RAND or McVerSi-Std.XO, we still found its performance to be notably inferior to McVerSi-ALL, and so do not include it in the final results in §6.6. In a similar vein, we do not include configurations without our simulator-specific optimizations in our results, as without these, the simulation runs were impracticably slow (around a couple of orders of magnitude slower).

Test Generation Parameters: Key parameters for all configurations are shown in Table 6.3. Note that these parameters were determined to give good results in a limited design space exploration. The *test size* of 1k operations is sufficient to find all studied bugs; in fact, we determine that larger test sizes cause performance to degrade, as the evolution of tests simply takes longer. We must ensure that tests are large enough to be able to detect most bugs in the first place, but not too large to limit the search performance. With this test size and depending on several factors (conflicting accesses, L1/L2 cache hits/misses, etc.), we note that the checker (§6.4.1) generally uses between 30% and 40% of the total wall-clock time.

The *test memory size* denotes the *usable* address range. The *stride* merely affects the base address (base addresses are generated in multiples of stride). In order to ensure cache capacity evictions take place, the test memory is *partitioned* in contiguous blocks of 512B, where the respective starting addresses of partitions are separated by a range of 1MB; e.g. in the case of 8KB, 16 such 512B partitions exist. As we are running full-system simulations, the allocation is not fully under our control, and the virtual memory manager (VMM) of the OS has final control over placement in physical memory. In our experiments, however, we observe our chosen test memory partitioning to have the desired effect.

The selected *operations and their bias*, while independent of a particular ISA, should be guided by the target MCM. In our case, to cover all enforced orderings of x86-TSO, the presented operations are sufficient. For more relaxed MCMs, the set of operations that need to be generated could be more extensive.

6.5.2.2 diy-litmus

The diy tool suite [Alg+12] automates litmus test generation, using knowledge of the MCM to generate a number of short tests which may trigger interesting behavior. Litmus tests are self-checking, i.e. they include the code for performing checking.

We generate all litmus tests for x86-TSO—we use all 38 tests available. We modified the run-script to exit the simulation on a detected MCM violation. As the simulation is time-limited (24 hours), and realistically it is not possible to pre-determine which of the litmus test will detect an error, we choose conservative parameters to limit the runtime of an individual litmus test, but re-execute all tests (in an outer loop) after the last of the tests has been executed. Thus the litmus tests may run until the simulation is terminated by the time-limit. For simulation we choose the following parameters: -st 4 (stride), -r 3 (runs), -s 8000 (size of test, iterations).

6.5.3 Selected Bugs

The following outlines the 11 studied bugs, 2 of which have not been discovered in Gem5 prior to this work. All bugs marked with a "*" denote real bugs in Gem5; others refer to artificially injected bugs. The prefix of the name we give a bug denotes which protocol is affected, as well as if either Load Queue (LQ) or Store Queue (SQ) contribute to the bug manifestation. We study two cache coherence protocols, one being the Ruby MESI implementation in Gem5, and the other the proposed TSO-CC (Chapter 4) protocol. TSO-CC provides an interesting case-study, as it implements a lazy consistency-directed coherence protocol for TSO. TSO-CC explicitly violates SWMR, a key invariant of traditional coherence protocols such as MESI, which makes it arguably more difficult to verify adherence to memory consistency using formal verification approaches such as model checking.

MESI,LQ+IS,Inv*: This bug causes read-read reordering (same or different addresses) that is prohibited by TSO. It is caused by the coherence protocol failing to forward an invalidation to the LQ after sinking an incoming Inv (invalidate) request in the IS (invalid-to-shared) transient state. The correct behavior would be to forward the invalidate along with the data once the data response message is received in the IS_I (invalid-to-shared, sunk invalidate) transient state. This is a real bug in Gem5, that had not been discovered previously. It is worth noting that this bug could not have been found via individual verification of either the coherence protocol (SWMR is not violated) or the LQ. The fix required correcting both components, and the Gem5

developers have been notified.

Note that this bug, as well as all following bugs with prefix MESI,LQ are variants of the "Peekaboo" problem [SHW11]—in these cases, arising due to speculative execution. Indeed, Gem5's implementation of the LQ provides correct behavior on a forwarded invalidation: if there exist any unperformed older reads and an invalidation is received, all newer reads are retried. However, if the coherence protocol never forwards an invalidation as is the case here, then newer reads may observe stale values.

MESI,LQ+SM,Inv*: This bug also causes read-read reordering (same or different addresses). It is caused by the coherence protocol failing to forward an invalidation to the LSQ in the SM (shared-to-modified) transient state upon receiving an Inv request. This bug has not been discovered previously. The fix only required correcting the coherence protocol, and a patch has been sent upstream to Gem5.

MESI,LQ+E,Inv: This bug results in read-read reordering (same or different addresses). It is caused by the coherence protocol failing to forward an invalidation to the LQ in the E state upon receiving an Inv.

MESI,LQ+M,Inv: Similar to MESI,LQ+E,Inv, but fails to forward an invalidation to the LQ in the M state.

MESI,LQ+S,Replacement: This bug is caused by the coherence protocol failing to forward an invalidation to the LQ upon replacement in the S state. It results in read-read reordering (same or different addresses).

MESI+PUTX-Race*: This bug is caused by a protocol race condition and subsequent invalid transition. It is described in detail by Komuravelli et al. [KAC14], who previously discovered it via model checking with Murφ. This bug does not manifest as an MCM bug directly, but instead is caught by Ruby as an invalid transition. If such a protocol had passed to a post-silicon stage, the effect the bug can have is not very clear: the result may be unexpected behavior (including an MCM bug) or something arguably more critical (e.g. system lockup). This bug has since been fixed in Gem5 (in January 2011).

MESI+Replace-Race: This bug is another protocol race; however, it is more subtle in nature. It manifests due to an L1 replacement in M and simultaneous L2 replacement of a previously clean block in MT (potentially modified, in local L1), where the L2 does not expect modified data, thereby failing to write back the modified block to memory.

TSO-CC+no-epoch-ids: To reset timestamps, TSO-CC requires epoch-ids to avoid races between timestamp-reset messages and read/write requests. Eliminating epoch-ids causes TSO violations (read-read reordering).

TSO-CC+compare: This bug is subtler than the previous one. In the presence of timestamp-groups, §4.2.3 states "where the requested line's timestamp is larger or equal than the last-seen timestamp from the writer of that line self-invalidate all Shared lines"—we change the comparison to just *larger than*. This bug causes read-read reordering.

LQ+no-TSO*: This bug causes read-read reordering to different addresses. The bug is caused by the LQ not squashing subsequent reads after an incoming forwarded invalidation from the coherence protocol. We previously discovered this bug via litmus testing, and sent a fix upstream in March 2014. This bug has also been independently discovered by PipeCheck [LPM14].

SQ+no-FIFO: This bug causes write-write reordering by not writing back in FIFO order, but instead out-of-order from the SQ.

6.6 Experimental Results

This section discusses the results we obtain for each individual test generation approach. First and foremost, we are interested in bug coverage, which addresses the bug-finding guarantees that each approach provides. This is followed by analysis of structural coverage, which addresses how thoroughly each approach explores the coherence protocol state transitions.

6.6.1 Bug Coverage

As seen in Table 6.4, the only configuration consistently finding all bugs in under 24 hours is our GP-based approach McVerSi-ALL (8KB). In comparison, McVerSi-RAND (best case with 1KB) only finds 8/11 of bugs and litmus tests only 2/11 bugs consistently within 24 hours. Furthermore, we can see that even when the competing approaches successfully find all bugs consistently within 24 hours, our GP-based approach almost always finds them sooner. This confirms our hypothesis that, although litmus testing and pseudo-random testing are effective post-silicon verification methodologies, without substantial optimizations, they are unsuitable for practical simulation-based verification.

What guarantees are provided with increasing runtime? Other than our McVerSi-ALL (8KB), no other configuration is able to find all bugs within 1 day. But what happens when the competing non-GP approaches (pseudo-random and litmus tests) are run for more than 1 day? Recall that we run each generator/bug pair 10 times

(samples) up to 24 hours.[5] Since the non-GP approaches are stateless (they do not continuously update their internal state to progress towards better tests), we note that running each bug 10 times for 24 hours is tantamount to measuring bug coverage when running for up to 10 days. Table 6.5 summarizes the results under this assumption. It is worth noting that neither litmus nor pseudo-random tests are able to find all bugs within effectively 10 days of running time. Although McVerSi-RAND (8KB) can guarantee finding additional bugs after running for more than 1 day, 2 out of 11 bugs (18%) are still not found. In these cases (NF in Table 6.4), the implication is that the test generator would either need more than 10 days, or is incapable of generating tests required to expose the particular bug.

How does usable address range affect test quality? With just 1KB of test memory, all test generation schemes achieve similar results. Because of the constrained address space, tests consist of a large number of conflicting accesses even if generated randomly. However, note that both GP approaches improve the average time to find all bugs over the pseudo-random test generator even with just 1KB of test memory; in particular, McVerSi-ALL reduces the average time by 27% in comparison with McVerSi-RAND. It is important to note, however, that none of the approaches using 1KB of test memory are able to find the following bugs: MESI,LQ+S,Replacement, MESI+PUTX-Race, and MESI+Replace-Race. Clearly, we require a larger test memory size to find these. With 8KB of test memory, McVerSi-ALL is able to find all bugs in all simulation runs, including the 3 bugs above.

How effective is our selective crossover? We note that McVerSi-Std.XO is unable to find certain bugs (NF), in cases where the bugs only manifest due to racy accesses. Those configurations not making use of the proposed selective crossover simply do not converge towards suitable MCM tests with high non-determinism/races; i.e. their set of candidate executions is too small to have a high probability of encountering the sequence of events in the system required to expose faulty logic. In order to find bugs which only manifest due to replacements, a large address range is required but also *suitable* MCM tests, i.e. highly racy tests. The bugs which are *only* found by McVerSi-ALL (8KB) require tests with an average ND_T of at least 2.0, and often greater than 3.0. The 1KB configurations' initial set of tests automatically achieve an average ND_T exceeding 2.0, whereas the 8KB configurations start out with an ND_T of around 1.1. At 8KB, only McVerSi-ALL is able to generate tests with an ND_T of 2.0 or above.

[5] For practical reasons, we are restricted to 24 hours per run.

Table 6.4: Bug coverage: *bug found count out of 10 samples (arith. mean hours to find the bug across 10 samples)*; NF = "Not Found within 24 hours"; **bold** highlights configurations which consistently find the bug within 24 hours.

Bug	*McVerSi-ALL (1KB)*	*McVerSi-ALL (8KB)*	*McVerSi-Std.XO (1KB)*	*McVerSi-Std.XO (8KB)*	*McVerSi-RAND (1KB)*	*McVerSi-RAND (8KB)*	*diy-litmus*
MESI,LQ+IS,Inv	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.73)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv	**10 (0.33)**	**10 (5.20)**	**10 (0.27)**	1 (5.01)	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (3.22)**	**10 (0.16)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (2.40)**	7 (3.80)	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replacement	NF	**10 (2.69)**	NF	4 (15.05)	NF	6 (10.10)	NF
MESI+PUTX-Race	NF	**10 (4.64)**	NF	5 (8.83)	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.50)**	NF	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	NF	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.02)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.83)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.80)	67 (2.31)	80 (0.97)	70 (3.41)	40 (3.60)

Table 6.5: Bugs found, when running up to the equivalent of 10 days time.

Bugs found within	1 day	5 days	10 days
McVerSi-ALL (8KB)	**100%**	N/A	N/A
McVerSi-RAND (1KB)	73%	73%	73%
McVerSi-RAND (8KB)	55%	73%	82%
diy-litmus	18%	45%	45%

Table 6.6: Maximum total *transition coverage* observed across all simulation runs.

Protocol	*McVerSi-ALL (1KB)*	*McVerSi-ALL (8KB)*	*McVerSi-Std.XO (1KB)*	*McVerSi-Std.XO (8KB)*	*McVerSi-RAND (1KB)*	*McVerSi-RAND (8KB)*	*diy-litmus*
MESI	60.9%	82.3%	62.3%	81.9%	60.9%	81.9%	66.5%
TSO-CC	51.8%	63.1%	50.8%	41.2%	51.8%	62.6%	54.8%

6.6.2 Structural Coverage

What is the impact of using coverage as fitness? Table 6.6 shows the maximum total achieved coverage (higher is better). Recall that, the fitness function we use does not make use of the total coverage, and instead focuses on rare transitions to avoid getting stuck in a local maximum. We note that the implementations of the variant of the MESI protocol and TSO-CC contain transitions which are extremely unlikely to occur (e.g. replacements in transient states from invalid—the LRU replacement policy in use is very unlikely to select such blocks), which we did not exclude from the coverage calculation, and therefore we do not reach 100%.

From Table 6.6 we can see that McVerSi-ALL (8KB) achieves highest coverage for both the MESI protocol and TSO-CC. Using coverage as fitness achieves its goal, leading to the improved performance (bug coverage discussed above) of McVerSi-ALL compared to McVerSi-RAND. More importantly, while the selective crossover continually increasing ND_T could have a negatively correlated effect on coverage, the GP-based approach ensures balance by simply proceeding to no longer select individuals with too high ND_T.

6.7 Related Work

This section provides a broader overview of methodologies for verifying that the coherence protocol, the memory system and other components of a system adhere to the MCM.

6.7.1 Formal Verification

While formal verification provides the highest possible guarantees, i.e. a proof of correctness, the model being verified against its specification is typically a component abstraction of what is present in the functional design implementation; with the coherence protocol being the main artifact being subjected to formal verification [PD97]. For most model checking approaches [ASL03; Che+06; CMP04; KAC14; McM01], the consistency properties intended to capture MCM correctness are derived properties, such as the SWMR [SHW11] invariant; these are inadequate for protocols explicitly violating such properties (e.g. lazy self-invalidation based protocols using "tear-off" blocks [LW95]). More powerful formal methods approaches for coherence protocol verification use operational models [CSG02; PD96; PD98; PD00], but require more user-effort to set up.

To raise confidence in a design, it would be prudent to *apply the best tools at each stage in the design*. Indeed, the recent model checking of the MESI coherence protocol of the GEMS memory simulator (and of Gem5) has found bugs [KAC14] (MESI+PUTX-Race among others). Independent of the memory system, PipeCheck [LPM14] can be used for model checking of pipeline abstractions (albeit against selected litmus tests), which also uncovered a bug in Gem5 (LQ+no-TSO). None of the above approaches can ensure the correctness of the interaction between components as we observed with several of the studied bugs (§6.5.3).

The recently published and concurrently developed CCICheck [Man+15] (based on PipeCheck [LPM14]), provides a methodology for verifying pipeline and memory system (with focus on coherence protocol) together. Broadly, their motivation is similar, in that the interaction between components is crucial in enforcing the consistency model, and unconventional protocols cannot easily be verified using traditional approaches (e.g. TSO-CC is also used as a case-study). By using abstract axiomatic models of the pipeline (like PipeCheck) and memory system, the result is exhaustive (on input litmus tests).

While extremely valuable at an early stage in the design, the above approaches are

only tractable with abstractions of the relevant parts of a full-system functional design, and thus are complementary to McVerSi.

6.7.2 Memory System Verification

Verifying the detailed implementation of the memory system in isolation can be done in simulation. Wood et al. [WGK90] present a methodology where a stub CPU takes control of the operations being issued to the memory system; tests are randomly selected operations from user "action/check scripts." A similar approach is taken by [Pon+98]. None of the approaches automatically feed back coverage metrics into the test generation.

The "witness string" method [ASL03; CAL04] generates test vectors for RTL simulation of the coherence protocol which cover distinct states, thereby improving test quality and reducing redundant simulation time. The witness strings are generated with the Murφ model checker, based on a model of the protocol. This approach, however, depends on an external tool and is not tightly coupled with the simulation tool.

In the presence of a detailed FSM of the coherence protocol, [WB08; QM12] propose methods to automatically inject events into the memory system to cover previously uncovered states and transitions. This requires detailed knowledge of the memory system's FSM, and in the absence of other control logic (e.g. core pipeline), is a feasible approach to generating high coverage. Yet, it would be much more difficult to accomplish in a full-system, where the test input to the system does not directly control the memory system, and instead is subject to other constraints of the control logic.

While these approaches target simulation, unlike them, McVerSi targets *full-system* simulation, and also demonstrates checking a complete axiomatic MCM.

6.7.3 Full-System Verification

Related work in this area has primarily focused on the problem of checker complexity due to limited visibility in a post-silicon environment. In the absence of conflict order visibility, checking an axiomatic MCM has been proven to be NP-complete [GK97].

To address the complexity of MCM checking in simulation, *relaxed scoreboards* (operational models) have been proposed [Sha+08; Sah+95] to monitor every memory operation's correct behavior. While this would even allow using real workloads and monitor the system on-the-fly, the test generation method is independent of the proposed

checking method. We find that checking an axiomatic MCM is fast enough for the relatively short tests used in our GP-based approach.

TSOtool [Han+04; MH05] and derivative algorithms [Roy+06; McL+15] propose approximate solution to the MCM checking problem in a post-silicon environment, due to the limited conflict order visibility. Hardware extensions to facilitate fast checking in post-silicon have been proposed, e.g. via counters [Che+09; Hu+12] or even re-partitioning of the cache to log ordering information [DWB09]. While throughput in post-silicon environments is generally higher, and therefore all use user constrained random tests, all of these approaches are only applicable at the very latest stages of a design.

Manually directed short tests, also called *litmus tests*, are also very common. More recently, diy [Alg+12] automates litmus test generation, using knowledge of the MCM to generate short tests which may trigger interesting behavior. Litmus tests have the advantage that they are self-checking, and therefore are more portable and simpler to set up for a wide variety of MCMs.

The above approaches target post-silicon testing, and none are specifically optimized for simulation.

6.7.4 Hardware Support for MCM Verification

An alternative approach to ensuring correctness is fault-tolerance via hardware support for detecting MCM violations dynamically and recovering from them [MS09; MQT12; Qia+13; RLS10]. Our proposal, which focusses on test generation, is orthogonal to these works. In particular, Romanescu et al. [RLS10] focus on detecting address translation (AT) related bugs with the help of their AT-aware MCM specifications. Our framework targets a full-system environment, including the TLB and MMU, and thus is also capable of detecting AT bugs (we did not detect any). Note, however, that our framework currently does not stress AT related aspects (it can handle synonyms, but does not explicitly deal with memory mapping operations, etc.), which we reserve for future work.

6.8 Conclusion

We have presented McVerSi, a test generation framework for fast memory consistency verification in full-system simulation. At later pre-silicon design stages, it is imperative

to rigorously verify the full-system. Due to the complexity of the implementation at this stage, verification methodologies with rigorous test generation are of great importance to raise the designer's confidence.

In the domain of simulation-based MCM verification, there is need for an approach which automatically improves test quality based on feedback from the simulation. This is a difficult search problem with many hidden variables, especially in multiprocessor systems, where the interleaving of threads is inherently non-deterministic. Indeed, the enforcement of an MCM is what brings order into the non-deterministic world of multiprocessors.

Our key contribution is a GP-based test generation approach, which generates effective MCM tests. By proposing a novel crossover which favors non-determinism, the generated tests increase the probability of the implementation having to work harder to enforce the required ordering guarantees of the MCM. Then, by using coverage as the fitness function, our approach evolves high-quality tests automatically. Our results show that, compared with alternative test generation approaches, we find all 11 considered bugs consistently, providing much higher guarantees about the classes of bugs McVerSi is capable of finding within practical time bounds. While it may be conceivable to achieve similar results via manual test generation, our approach automatically explores tests satisfying the coverage criteria without user intervention.

The framework we present offers the building blocks for researchers and industrial designers alike, to evaluate coherence protocols and other microarchitectural artifacts to adhere to the promised consistency model early in the design cycle. A simulator-independent C++ library (including consistency model descriptions, checker, and test generator) is provided online: http://ac.marcoelver.com/research/mcversi

Part V

Conclusions

Chapter 7

Conclusions and Future Directions

7.1 Opening Pandora's Box

This thesis has explored aspects of memory consistency directed cache coherence protocol design to overcome some of the scaling challenges—particularly on-chip storage overheads—with increasing number of processors in multiprocessors. This will become even more important if designers wish to increase the number of cores in CMPs ("multicores") in the foreseeable future.

In particular, this thesis highlights the fact that not just consistency models exposing synchronization information explicitly (e.g. RC) are amenable to a lazy coherence approach. Indeed, stricter models can also benefit from the advantages of a lazy coherence approach, in particular, reduced on-chip storage overheads. We demonstrate this by first proposing TSO-CC (Chapter 4), a *lazy coherence protocol for TSO without the overhead of maintaining lists of sharers*. Furthermore, we propose *transitive reduction (using timestamps) of acquires* in the absence of explicit synchronization to realize performance on par with a conventional MESI baseline protocol.

Next, we show that TSO-CC's transitive reduction optimization together with explicit synchronization can provide greater storage savings by proposing RC3 (Chapter 5), a lazy coherence protocol for RCtso. Especially modern programming language consistency models already provide the explicit synchronization information required, something that the hardware should ideally exploit via the RC3 protocol. Unfortunately, an architecture such as x86-64 guaranteeing TSO does not allow conveying this information to the hardware. While TSO-CC can be employed without further changes, we also demonstrate how to employ RC3 on x86-64 via a backward compatible ISA extension changing the consistency model from TSO to RCtso. With further reduced

storage overheads, RC3 provides performance on par with TSO-CC as well as good performance for most legacy codes that still assume TSO.

Several previous lazy coherence protocol approaches [Cho+11; RK12] also argue that this approach affords simpler protocols relative to conventional eager protocols. We initially conjectured simplicity to be another argument for the lazy coherence approaches proposed in this thesis. This, however, may no longer be the case for the consistency models and real-world architectural constraints targeted (both TSO-CC and RC3 are evaluated using full-system simulation).

The quote by Sorin, Hill, and Wood [SHW11]—"opening the coherence box incurs considerable intellectual and verification complexity, bringing to mind the Greek myth about Pandora's box"—did provide a hint of the challenges, but also research opportunities of this approach. Indeed, consistency directed coherence protocols cannot use conventional coherence definitions (e.g. SWMR) to be verified against, and few existing verification methodologies apply. Furthermore, as the full consistency model is used as a specification, the interaction with other components of a system must not be neglected. Out of these realizations, we propose a novel methodology for rigorous simulation-based verification.

McVerSi (Chapter 6) is unique in its ability to generate *high-quality test cases for fast memory consistency verification of a detailed full-system* implementation in simulation, something we felt none of the existing verification approaches could provide. The results show that the McVerSi approach finds bugs faster compared to alternative approaches such as pseudo-random or litmus tests (with some bugs only found by McVerSi). An approach like McVerSi is not just valuable for verifying consistency directed protocols in the context of a full-system, *but also conventional eager protocols*. This is highlighted by the fact, that we discovered 2 new bugs in the MESI protocol provided by the Gem5 simulator—bugs caused by the faulty interaction between protocol and pipeline.

The protocols and verification method proposed in this thesis demonstrates not only the potential of consistency directed cache coherence protocol design but also how to manage some of the new challenges associated with this approach. As designers are looking for ways to increase core counts, consistency directed designs are one option to achieve this—like in may other areas, specialization can provide the means to a suitable solution.

7.2 Critical Analysis

This section provides a brief critical analysis of the work in this thesis, focusing on perspectives suitable for future work.

7.2.1 Cache and Directory Organization

As mentioned in §4.2.8, for the protocols evaluated, a simple cache and directory organization was chosen. First, as the focus of this thesis was on designing consistency directed protocols, rather than optimize organization, this was to highlight that even in the presence of a simple organization we can achieve substantial savings. Second, this choice was pragmatic, in order to avoid introducing further complexities that would distract from the main theme of this thesis.

However, as suggested in §4.2.8, combining the proposed lazy coherence protocols (TSO-CC and RC3) with approaches optimizing directory organization and data structures would result in even greater storage savings. Some approaches would be trivial to apply (e.g. [Fer+11]), but others may require modification and some thought about more low-level aspects of an implementation. The overall scalability of a combined approach should be even greater. This would also allow for a fairer comparison with related work that focus on cache and directory organization.

7.2.2 Conversion to RCtso

In the evaluation of RC3 (§5.5) is performed in the context of a full-system with workloads converted to use the x86-RCtso consistency model. Although GCC was modified to emit the modified instructions, some parts of the software stack may not have been converted optimally. In particular, some synchronization libraries that rely on hand-written assembly code, will not have been translated optimally. Furthermore, the Linux kernel used was not modified; here, it became apparent that a proper conversion was well beyond the scope of this thesis.

We, therefore, assume that the performance results presented for RC3 are likely conservative. In practice, if a rigorous conversion would be performed, overall performance should be better than the presented results.

7.2.3 Transparency of Genetic Programming

Although the use of genetic programming for McVerSi (Chapter 6) can be justified due to the complexity of the problem—the only inputs are the test programs and the control over and information about the system is otherwise limited—precisely understanding why tests evolve a certain way is difficult. Indeed, the sentiment was shared by one of the anonymous reviewers of the McVerSi paper:

> *In general, I am not a fan of evolutionary approaches, especially genetic algorithms, because it is not easy to figure out why they work. However, I must admit that the results of this paper are so good that it'd be a pity to not have it on the HPCA program.*

7.3 Future Directions

To conclude, this section discusses broader open questions and research opportunities beyond this thesis.

7.3.1 Microarchitectural Gaps and Power Modelling

Several open questions regarding the low-level microarchitectural implementation of some cache operations and structures (self-invalidations, timestamp operations, etc.) should be answered. In particular, lazy coherence protocols rely on selective self-invalidations of cache lines. Several options for implementations of self-invalidations are mentioned by Lebeck and Wood [LW95], but their respective details are not analyzed further. This may be a good starting point for further study, in particular, the realization of a detailed analytical power model.

This is required to also achieve faithful power modeling of systems with lazy coherence protocols—existing efforts have only been partial, estimating this portion to be negligible [RK12; KK10]. Finally, the realization of an RTL prototype with a lazy coherence protocol would answer many of these open questions.

7.3.2 Better Formal Verification

In order to manage more complex consistency directed protocols, automated formal verification methods are necessary to complement simulation-based approaches like McVerSi (which is not exhaustive yet rigorous, to handle much more detailed imple-

mentations). In particular, tools are required that can handle the types or protocols (lazy protocols) and specifications (consistency model) proposed in this thesis.

The current state of the art (§6.7) is not there yet, something we realized while we were attempting a parameterized proof of the optimized TSO-CC protocol via model checking. Furthermore, approaches that provide a proof that the protocol, together with given constraints of how it interacts with other components (e.g. an out-of-order pipeline), guarantees the promised consistency model would be very valuable. The concurrently developed CCICheck [Man+15] is a step in the right direction, but its proof (over litmus tests only) as well as modeling capabilities (e.g. TSO-CC is modeled without timestamps, as difficult to express axiomatically) are limited. Indeed, providing an automated proof of correctness of even abstract models with detail beyond manual reasoning are extremely valuable. Automating as much of this process as possible will be vital to be of use to the wider community.

Part VI

Appendix

Appendix A

Detailed Protocol Specification for TSO-CC

This chapter provides a detailed specification of the TSO-CC protocol (Chapter 4). It is intended as a precise description to be used as a reference for an implementation. The below uses literals introduced in Table 4.1.

A.1 Assumptions and Definitions

1. The protocol requires distinguishing valid and invalid timestamps (b.ts). In the following specification $\varnothing$ is used to denote an invalid entry. In our implementation, we use 0 to denote an invalid timestamp, which means the smallest valid timestamp is 1.
2. DataS, DataX and Data messages are expected to carry data.
3. A receive message action is of the format: source?Message.
4. A send message action is of the format: destination!Message.
5. A batch transition of all lines in states State1, State2, ... to state NextState is abbreviated tr_all {State1, State2, ...} NextState.

A.2 Protocol State Transition Tables

The state transition tables can be found in Tables A.1 and A.2. The following sections provide notes about events in the Tables marked by the respective raised number.

A.2.1 Private Cache Controller

1. We can't just set the state to Invalid, as the directory might have gotten a read and forwarded the request to us. So we must write back, and wait for Ack to ensure that the line propagated to the L2, and thus no more Fwd requests are outstanding.

2. If $B_{write-group} = 0$, in the presence of non-infinite timestamps, the comparison operator cannot be $<$, as it would violate correctness. This is due to how timestamp resets are dealt with in the L2 (see §A.3.3).

3. Must reset timestamp, in case the line has since been evicted from L2 and we obtain it in Exclusive. If this is the case, the line may have been modified by another node; now, if we get a FwdX request, the old timestamp must not be forwarded.

A.2.2 Directory Controller

1. Reuse the block's b.owner bits to maintain a superset of SharedRO sharers: each bit is a pointer to $\lceil \frac{C_{L1}}{\lceil log(C_{L1}) \rceil} \rceil$ sharers.

2. Checking if a line's timestamp in the L2 is **decayed**. In order to allow Shared blocks which have not been written to in a long time to transition to SharedRO, we can use the timestamp *b.ts* and compare against the owner's entry in the last-seen table: check if a fixed period has passed between the last-seen timestamp and when the line was updated according to *b.ts*.

$$\mathsf{ts\_L1[b.owner]} > 2^{B_{ts}-n} \wedge \mathsf{b.ts} \leq \mathsf{ts\_L1[b.owner]} - 2^{B_{ts}-n}$$

Table A.1: TSO-CC private (L1) cache controller transition table.

	Read	Write	Evict	src?DataS(state, owner, ts)	src?DataX(owner, ts, ackc)	src?FwdS(dst)	src?FwdX(dst)	src?Ack	src?InvRO
Invalid	dir!GetS; b.ts ← ∅; → WaitS;	dir!GetX; update b.ts; → WaitX;							dir!AckRO;

Exclusive	hit;	hit;
		update b.ts;
		→ Modified;
			dir!PutE;
			→ WaitEI;[1]
						dst!DataS(SharedRO, self, b.ts);
						dir!Ack(0);
						→ SharedRO;
							dst!DataX(self, b.ts, 1);
							→ Shared;
									dir!AckRO;

Modified	hit;	hit;
		update b.ts;
			dir!Data(b.ts);
			→ WaitMI;
						dst!DataS(Shared, self, b.ts);
						dir!Data(b.ts);
						→ Shared;
							dst!DataX(self, b.ts, 1);
							→ Shared;
									dir!AckRO;

Shared	**if** b.acnt < maxac **then**
	increment b.acnt;
	hit;
	else
	dir!GetS;
	b.ts ← ∅;[3]
	→ WaitS;
	endif
		dir!GetX;
		update b.ts;
		→ WaitX;
			→ Invalid;						dir!AckRO;

SharedRO	hit;	update b.ts; dir!GetX; → WaitX;	→ Invalid;						dir!AckRO; → Invalid;

WaitS	stall;	stall;	stall;	copy_data; hit;
				reset b.acnt;
				if state = Exclusive **then**
				dir!Ack(0);
				endif
				→ state;
									dir!AckRO;
									→ WaitSROI;

WaitSROI	stall;	stall;	stall;	copy_data; hit;
				reset b.acnt;
				if state = Exclusive **then**
				dir!Ack(0);
				elif state = SharedRO **then**
				→ Invalid;
				endif
				→ state;
									dir!AckRO;

WaitX	stall;	stall;	stall;		copy_data; hit; reset b.acnt; dir!Ack(ackc); → Modified;				dir!AckRO;
WaitEI	stall;	stall;	stall;			dst!DataS(SharedRO, self, b.ts); → Invalid;	dst!DataX(self, b.ts, 0); → Invalid;	→ Invalid;	dir!AckRO;
WaitMI	stall;	stall;	stall;			dst!DataS(Shared, self, b.ts); → Invalid;	dst!DataX(self, b.ts, 0); → Invalid;	→ Invalid;	dir!AckRO;

DataS
@ WaitS, WaitSROI
DataX
@ WaitX
	if owner = ∅ ∧ ts ≠ ∅ **then**
	if ts_L2[src] < ts **then**
	ts_L2[src] ← ts;
	tr_all {Shared} Invalid;
	endif
	...
		...
		elif owner ≠ self ∧ (ts = ∅ ∨ ts_L1[owner] ≤ ts) **then**[2]
		if ts ≠ ∅ **then**
		ts_L1[owner] ← ts;
		endif
		tr_all {Shared} Invalid;
		endif

Table A.2: Directory (L2) controller transition table.

	p?GetS	p?GetX	p?Data(ts)	p?Ack(c)	p?PutE	p?AckRO
Invalid	p!DataS(Exclusive, ∅, ∅); b.owner ← p; b.ts ← ∅; → WaitE1;	p!DataX(∅, ∅, 0); b.owner ← p; b.ts ← ∅; → WaitE1;				
Uncached	p!DataS(Exclusive, b.owner, b.ts); b.owner ← p; b.ts ← ∅; → WaitE1;	p!DataX(b.owner, b.ts, 0); b.owner ← p; b.ts ← ∅; → WaitE1;				
Exclusive	b.owner!FwdS(p); tbe.sharers ← {p}; → WaitS;	b.owner!FwdX(p); b.owner ← p; b.ts ← ∅; → WaitE2;	copy_data; p!Ack; b.ts ← ts; → Uncached;		p!Ack; → Uncached;	

Shared	**if** expired b.ts ∨ decayed b.ts **then**[2]
	  b.owner ← {p};
	  update b.ts;
	  p!DataS(SharedRO, ∅, b.ts);
	  → SharedRO;
	else
	  p!DataS(Shared, b.owner, b.ts);
	endif
		p!DataX(b.owner, b.ts, 0);
		b.owner ← p;
		b.ts ← ∅;
		→ WaitE1;				

SharedRO	p!DataS(SharedRO, ∅, b.ts); b.owner ← b.owner ∪ {p};[1]	dst ← {q | $q \in$ b.owner ∧ $q \neq$ p}; dst!InvRO; tbe.need_acks ← |dst|; b.owner ← p; → WaitEn;				

WaitE1	stall;	stall;	**if** p ≠ b.owner **then**
			  → Exclusive;
			else
			  copy_data;
			  p!Ack;
			  b.ts ← ts;
			  → WaitU1;
			endif
				→ Exclusive;	**if** p ≠ b.owner **then**
					  → Exclusive;
					else
					  p!Ack;
					  → WaitU1;
					endif	

WaitE2	stall;	stall;	**if** p ≠ b.owner **then**
			  → WaitE1;
			else
			  copy_data;
			  p!Ack;
			  b.ts ← ts;
			  → WaitU2;
			endif
				if c = 1 **then**
				  → Exclusive;
				else
				  → WaitE1;
				endif
					if p ≠ b.owner **then**
					  → WaitE1;
					else
					  p!Ack;
					  → WaitU2;
					endif	

WaitU1	stall;	stall;	→ Uncached;	→ Uncached;	→ Uncached;	

WaitU2	stall;	stall;	→ WaitU1;	**if** c = 1 **then**
				  → Uncached;
				else
				  → WaitU1;
				endif
					→ WaitU1;	

WaitEn	stall;	stall;				tbe.need_acks − −;
						if tbe.need_acks = 0 **then**
						  b.owner!DataX(∅, b.ts, 0);
						  b.ts ← ∅;
						  → WaitE1;
						endif

WaitS	stall;	stall;	copy_data; b.ts ← ts; → Shared;	b.owner ← tbe.sharers ∪ {p}; update b.ts; → SharedRO;	b.owner ← tbe.sharers; update b.ts; → SharedRO;	

A.3 Additional Rules and Optimizations

The following is a list of additional rules and optimizations, which have an impact on both L1 and L2 controllers; this completes the full protocol description.

A.3.1 Cache Inclusivity and Evictions

Evictions from the L2 are omitted from the transition table; the following must hold: upon eviction of lines from the L2, inclusivity must be maintained for lines which are tracked by the L2 (Exclusive and SharedRO).

A.3.2 Timestamp Table Size Relaxations

The L1's timestamp-tables ts_L1 and ts_L2 do not need to be able to hold as many entries as there are respective nodes. Applying an eviction policy to evict entries from the timestamp-tables allows to have a reduced-size timestamp-table.

A.3.3 Effect of L1 Timestamp update

To **update** a timestamp in the L1 means assigning the locally maintained timestamp to the line, and also increment this timestamp based on either of the following policies:

1. Always (write-group $= 1$).
2. Write-groups: If constant number of writes falling under the same timestamp reached.

Timestamp overflows in the L1 are dealt with sending out a TimestampReset broadcast to L1s and L2 tiles:

1. Each L1 invalidates ts_L1[src] on receiving a TimestampReset.
2. Every L2 tile must also maintain a table ts_L1 of last-seen timestamps; ts_L1[src] is updated on every b.ts $\leftarrow$ ts, if ts is newer than the existing last-seen timestamp entry from an L1; on receiving a TimestampReset the respective entry is invalidated. The table of last-seen timestamps must be able to hold, unlike the L1's timestamp-tables, the full list of timestamps of every possible L1.
3. The L2 will assign a response message b.ts if the *last-seen timestamp from the owner is larger or equal to the line's timestamp* (not **expired**), *the smallest valid*

timestamp ($\varnothing$ is valid, but degrades performance) otherwise. Similarly for **all** L1 data messages by comparing against L1's own timestamp.

A.3.4 Effect of L2 Timestamp update

To **update** a timestamp in the L2 (for SharedRO) means assigning the L2-local timestamp to the line and incrementing the timestamp under the following conditions:

- *from*-Invalid, check against in WaitS to SharedRO transition: after an L2 eviction of a dirty line; after a GetS event in Uncached where b.ts $\neq \varnothing$ before resetting b.ts.
- *from*-Shared, check against in Shared to SharedRO transition: after a block transitions to Shared.

Maintain a bit for each from-condition to signify if the timestamp should be incremented on the next update or not, resetting *all* bits after the increment was performed.

In essence, the L2's timestamp should always be increment after a transition which can lead to a block ending up in the SharedRO state, but need not actually be incremented until the first block transitions to SharedRO.

It is possible to use only one bit for all conditions, but this would cause unnecessary timestamp increments when a cache line transitions to SharedRO based on the not-modified rule, as transitions to Shared are quite common, but Shared to SharedRO may not be, therefore it makes sense to maintain extra bits for each observed transition that may lead back to a SharedRO state, but only check the condition at the appropriate nearest transition to SharedRO.

Rule for when to increment an L2-timestamp: if a set of writes W happened before a set of transitions T that can cause likely transitions R to SharedRO, but we can not keep track of which blocks are affected, the system should remember that T happened so that upon the first transition in R we can allow L1s to deduce W happened before R. For two timestamps t and t', if $t < t'$ then $\mathsf{W} \rightarrow \mathsf{T} \rightarrow \mathsf{R} \rightarrow \mathsf{W}' \rightarrow \mathsf{T}' \rightarrow \mathsf{R}'$; in order to make visible all writes from W' the L1 needs to self-invalidate on R', if the largest timestamp value from the L2 it has seen is only t.

Dealing with timestamp overflows:

1. Timestamp overflows in the L2 are dealt with sending out a TimestampReset broadcast and each L1 resetting ts_L2[src]. To not send invalid timestamps, like

in §A.3.3, the L2 will assign a response message b.ts if the *current L2-local timestamp is larger or equal to the line's timestamp*, *the smallest valid timestamp* ($\varnothing$ is valid, but degrades performance) otherwise; in case the smallest valid timestamp is used, the next timestamp assigned to a line after an overflow must always be larger than the smallest valid timestamp.

2. In a NUCA architecture, it will be necessary to either propagate all increments to the L2-local timestamp across all tiles, or each L2 tile maintains its own timestamp, and the L1s maintain a ts_L2 entry per tile or cluster of tiles in a separate table.

A.3.5 TimestampReset Races

To resolve TimestampReset races, without requiring the sender of a TimestampReset to wait for acks, if we can assume a bounded time on message propagation delay:

1. Every node in the system (L1s and L2 tiles) maintains an epoch_id, which is set to a value different than the previous value on sending a TimestampReset; the TimestampReset message contains the new epoch_id. The number of bits required per epoch_id must be large enough to eliminate the probability of having more than one TimestampReset message with the same epoch_id in-flight, but small enough to satisfy storage requirements.

2. The L1s maintain a table of epoch_ids with entries for every L1 and L2 tiles in the system.

3. The L2 tiles each maintain a table of epoch_ids with entries for every L1.

4. On receiving a TimestampReset message, the sender's entry in the respective timestamp-table is invalidated but the epoch_ids entry for the sender is updated with the epoch_id that was received along with the TimestampReset message.

5. An epoch_id is sent with every Data, DataS and DataX message:

   - If the message originates from an L1, it is the L1s own epoch_id.
   - If the message originates from the L2, and owner $\neq \varnothing$, the entry in epoch_ids_L1[b.owner] is assigned.
   - If the message originates from the L2, and owner $= \varnothing \wedge$ ts $\neq \varnothing$, the L2's epoch_id is assigned.

6. The L2 updates epoch_ids_L1$[p]$ along with every b.ts $\leftarrow ts$. If epoch_ids_L1$[p] \neq$ epoch_id, the last-seen entry in ts_L1 must be updated (timestamp reset).

7. On receiving a DataS or DataX message, before the check for potential acquires, the L1 must perform the following check:

```
if ts ≠ ∅ ∧ owner ≠ self then
  if owner ≠ ∅ then
    if epoch_ids_L1[owner] ≠ msg_epoch_id then
      invalidate ts_L1[owner];
      epoch_ids_L1[owner] ← msg_epoch_id;
    endif
  elif epoch_ids_L2[src] ≠ msg_epoch_id then
    invalidate ts_L2[src];
    epoch_ids_L2[src] ← msg_epoch_id;
  endif
endif
```

If a self-invalidation is possible due to seeing a newer value than in the timestamp-tables *ts_L1* or *ts_L2* respectively, but not having done this check yet, check if the currently held epoch-id for the line's source is valid or not, if not, invalidate the entry in the timestamp-table, essentially performing the same action if a TimestampReset is received.

Bibliography

[ADC11] T. J. Ashby, P. Diaz, and M. Cintra. “Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters”. In: *IEEE Trans. Computers* 60.4 (2011), pp. 472–483. DOI: 10.1109/TC.2010.155.

[Adv13] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions*. Revision 3.20. May 2013.

[Adv93] S. V. Adve. “Designing Memory Consistency Models For Shared-Memory Multiprocessors”. PhD thesis. 1993. URL: http://rsim.cs.uiuc.edu/~sadve/Publications/thesis.pdf.

[AG96] S. V. Adve and K. Gharachorloo. “Shared Memory Consistency Models: A Tutorial”. In: *IEEE Computer* 29.12 (1996), pp. 66–76. DOI: 10.1109/2.546611.

[Aga+09] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. “GARNET: A detailed on-chip network model inside a full-system simulator”. In: *ISPASS*. 2009, pp. 33–42. DOI: 10.1109/ISPASS.2009.4919636.

[Aga+88] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. “An Evaluation of Directory Schemes for Cache Coherence”. In: *International Symposium on Computer Architecture (ISCA)*. 1988, pp. 280–289. DOI: 10.1109/ISCA.1988.5238.

[AH90] S. V. Adve and M. D. Hill. “Weak Ordering - A New Definition”. In: *International Symposium on Computer Architecture (ISCA)*. 1990, pp. 2–14. DOI: 10.1145/325164.325100.

[AH93] S. V. Adve and M. D. Hill. “A Unified Formalization of Four Shared-Memory Models”. In: *IEEE Trans. Parallel Distrib. Syst.* 4.6 (1993), pp. 613–624. DOI: 10.1109/71.242161.

[Aha+95] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. “Causal Memory: Definitions, Implementation, and Programming”. In: *Distributed Computing* 9.1 (1995), pp. 37–49. DOI: 10.1007/BF01784241.

[AHJ91] M. Ahamad, P. W. Hutto, and R. John. “Implementing and programming causal distributed shared memory”. In: *International Conference on Distributed Computing Systems (ICDCS)*. 1991, pp. 274–281. DOI: 10.1109/ICDCS.1991.148677.

[Alg+11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. “Litmus: Running Tests against Hardware”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2011, pp. 41–44. DOI: 10.1007/978-3-642-19835-9_5.

[Alg+12] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. “Fences in weak memory models (extended version)”. In: *Formal Methods in System Design* 40.2 (2012), pp. 170–205. DOI: 10.1007/s10703-011-0135-z.

[Alg+15] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. “GPU concurrency: Weak behaviours and programming assumptions”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015. DOI: 10.1145/2694344.2694391.

[AMT14] J. Alglave, L. Maranget, and M. Tautschnig. “Herding cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”. In: *ACM Trans. Program. Lang. Syst.* (2014). DOI: 10.1145/2627752.

[ARM14] ARM Limited. *ARMv8-A Reference Manual*. 2014.

[AS85] B. Alpern and F. B. Schneider. “Defining Liveness”. In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0.

[ASL03] D. Abts, S. Scott, and D. J. Lilja. “So Many States, So Little Time: Verifying Memory Coherence in the Cray X1”. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2003, p. 11. DOI: 10.1109/IPDPS.2003.1213087.

[BA08] H.-J. Boehm and S. V. Adve. “Foundations of the C++ concurrency memory model”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2008, pp. 68–78. DOI: 10.1145/1375581.1375591.

[Ban+98] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[Bau78] G. M. Baudet. "Asynchronous Iterative Methods for Multiprocessors". In: *J. ACM* 25.2 (1978), pp. 226–244. DOI: 10.1145/322063.322067.

[BFM09] N. Barrow-Williams, C. Fensch, and S. W. Moore. "A communication characterisation of Splash-2 and Parsec". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 86–97. DOI: 10.1109/IISWC.2009.5306792.

[Bie+08] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC benchmark suite: characterization and architectural implications". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 72–81. DOI: 10.1145/1454115.1454128.

[Bin+11] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The gem5 simulator". In: *SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[Bos+01] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. "A genetic approach to automatic bias generation for biased random instruction generation". In: *CEC*. 2001. DOI: 10.1109/CEC.2001.934425.

[CAL04] Y. Chen, D. Abts, and D. J. Lilja. "State Pruning for Test Vector Generation for a Multiprocessor Cache Coherence Protocol". In: *15th IEEE International Workshop on Rapid System Prototyping*. 2004, pp. 74–77. DOI: 10.1109/RSP.2004.40.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. "Implementation and Performance of Munin". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 1991, pp. 152–164. DOI: 10.1145/121132.121159.

[CCS03] F. Corno, F. Cumani, and G. Squillero. "Exploiting Auto-adaptive μGP for Highly Effective Test Programs Generation". In: *ICES*. 2003, pp. 262–273. DOI: 10.1007/3-540-36553-2_24.

[CF78] L. M. Censier and P. Feautrier. “A New Solution to Coherence Problems in Multicache Systems”. In: *IEEE Trans. Computers* 27.12 (1978), pp. 1112–1118. DOI: 10.1109/TC.1978.1675013.

[Che+06] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. “Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee”. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2006, pp. 81–88. DOI: 10.1109/FMCAD.2006.28.

[Che+09] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan. “Fast complete memory consistency verification”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2009, pp. 381–392. DOI: 10.1109/HPCA.2009.4798276.

[Cho+11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. “DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2011, pp. 155–166. DOI: 10.1109/PACT.2011.21.

[CMP04] C.-T. Chou, P. K. Mannava, and S. Park. “A Simple Method for Parameterized Verification of Cache Coherence Protocols”. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2004, pp. 382–398. DOI: 10.1007/978-3-540-30494-4_27.

[CSG02] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. “Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking”. In: *International Conference on Computer Aided Verification (CAV)*. 2002, pp. 123–136. DOI: 10.1007/3-540-45657-0_10.

[Cue+11] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. “Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks”. In: *International Symposium on Computer Architecture (ISCA)*. 2011, pp. 93–104. DOI: 10.1145/2000064.2000076.

[Den+74] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *Solid-State Circuits, IEEE Journal of* 9.5 (Oct. 1974), pp. 256–268.

[Dil96] D. L. Dill. “The Mur*phi* Verification System”. In: *International Conference on Computer Aided Verification (CAV)*. 1996, pp. 390–393. DOI: 10.1007/3-540-61474-5_86.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. “Memory Access Buffering in Multiprocessors”. In: *International Symposium on Computer Architecture (ISCA)*. 1986, pp. 434–442. DOI: 10.1145/17407.17406.

[DSS10] L. Dalessandro, M. F. Spear, and M. L. Scott. “NOrec: streamlining STM by abolishing ownership records”. In: *PPOPP*. 2010, pp. 67–78. DOI: 10.1145/1693453.1693464.

[Dub+91] M. Dubois, J.-C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. “Delayed consistency and its effects on the miss rate of parallel programs”. In: *ACM/IEEE Conference on Supercomputing (SC)*. 1991, pp. 197–206. DOI: 10.1145/125826.125941.

[DWB09] A. DeOrio, I. Wagner, and V. Bertacco. “Dacota: Post-silicon validation of the memory subsystem in multi-core designs”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2009, pp. 405–416. DOI: 10.1109/HPCA.2009.4798278.

[Esm+11] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark silicon and the end of multicore scaling”. In: *International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[FC08] C. Fensch and M. Cintra. “An OS-based alternative to full hardware coherence on tiled CMPs”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2008, pp. 355–366. DOI: 10.1109/HPCA.2008.4658652.

[Fer+11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. “Cuckoo directory: A scalable directory for many-core systems”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2011, pp. 169–180. DOI: 10.1109/HPCA.2011.5749726.

[Fer+12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”. In: *International Conference on Architectural Support for Program-*

ming Languages and Operating Systems (ASPLOS). 2012. DOI: 10.1145/2150976.2150982.

[Flu+16] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2016. URL: http://www.cl.cam.ac.uk/~pes20/popl16-armv8/top.pdf.

[GGH91] K. Gharachorloo, A. Gupta, and J. L. Hennessy. "Two Techniques to Enhance the Performance of Memory Consistency Models". In: *International Conference on Parallel Processing (ICPP)*. 1991, pp. 355–364.

[Gha+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". In: *International Symposium on Computer Architecture (ISCA)*. 1990, pp. 15–26. DOI: 10.1145/325164.325102.

[Gha95] K. Gharachorloo. *Memory Consistency Models For Shared-Memory Multiprocessors*. Tech. rep. CSL-TR-95-685. 1995. URL: http://i.stanford.edu/pub/cstr/reports/csl/tr/95/685/CSL-TR-95-685.pdf.

[GK97] P. B. Gibbons and E. Korach. "Testing Shared Memories". In: *SIAM J. Comput.* 26.4 (1997), pp. 1208–1244. DOI: 10.1137/S0097539794279614.

[GL96] D. B. Gustavson and Q. Li. "The Scalable Coherent Interface (SCI)". In: *IEEE Communications Magazine* 34.8 (1996), pp. 52–63. DOI: 10.1109/35.533919.

[GWM90] A. Gupta, W.-D. Weber, and T. C. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes". In: *International Conference on Parallel Processing (ICPP)*. 1990, pp. 312–321.

[Han+04] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model". In: *International Symposium on Computer Architecture (ISCA)*. 2004, pp. 114–123. DOI: 10.1109/ISCA.2004.1310768.

[Hec+14] 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: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2014. DOI: 10.1109/HPCA.2014.6835930.

[Hie+09] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan. “Using formal specifications to support testing”. In: *ACM Comput. Surv.* 41.2 (2009). DOI: 10.1145/1459352.1459354.

[Hil+92] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. “Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocesors”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1992, pp. 262–273. DOI: 10.1145/143365.143537.

[HMK02] Z. Hu, M. Martonosi, and S. Kaxiras. “Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior”. In: *International Symposium on Computer Architecture (ISCA)*. 2002, pp. 209–220. DOI: 10.1109/ISCA.2002.1003579.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

[How+14] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. “Heterogeneous-race-free memory models”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014, pp. 427–440. URL: http://research.cs.wisc.edu/multifacet/papers/asplos14_hrf_updated.pdf.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)* Morgan Kaufmann, 2007.

[Hu+12] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li. “Linear Time Memory Consistency Verification”. In: *IEEE Trans. Computers* 61.4 (2012), pp. 502–516. DOI: 10.1109/TC.2011.41.

[IE12] C. Ioannides and K. Eder. “Coverage-Directed Test Generation Automated by Machine Learning - A Review”. In: *ACM Trans. Design Autom. Electr. Syst.* 17.1 (2012), p. 7. DOI: 10.1145/2071356.2071363.

[Int02] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. 2002.

[Int14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Feb. 2014.

[ISO11a] ISO/IEC. *Programming Languages — C*. ISO/IEC 9899:2011. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf. 2011.

[ISO11b] ISO/IEC. *Programming Languages — C++*. ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf. 2011.

[Jam+90] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. “Distributed-directory scheme: scalable coherent interface”. In: *IEEE Computer* 23.6 (1990), pp. 74–77. DOI: 10.1109/2.55503.

[Jos+03] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. “Checking Cache-Coherence Protocols with TLA$^+$”. In: *Formal Methods in System Design* 22.2 (2003), pp. 125–131. DOI: 10.1023/A:1022969405325.

[KAC14] R. Komuravelli, S. V. Adve, and C.-T. Chou. “Revisiting the Complexity of Hardware Cache Coherence and Some Implications”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).

[KBK02] C. Kim, D. Burger, and S. W. Keckler. “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2002, pp. 211–222. DOI: 10.1145/605397.605420.

[KCZ92] P. J. Keleher, A. L. Cox, and W. Zwaenepoel. “Lazy Release Consistency for Software Distributed Shared Memory”. In: *International Symposium on Computer Architecture (ISCA)*. 1992, pp. 13–21. DOI: 10.1145/139669.139676.

[Kel+10] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel. “WAYPOINT: scaling coherence to thousand-core architectures”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010, pp. 99–110. DOI: 10.1145/1854273.1854291.

[Kel+94] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems”. In: *USENIX Winter*. 1994, pp. 115–132.

[KK10] S. Kaxiras and G. Keramidas. “SARC Coherence: Scaling Directory Cache Coherence in Performance and Power”. In: *IEEE Micro* 30.5 (2010), pp. 54–65. DOI: 10.1109/MM.2010.82.

[Koz92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[KSB95] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. “Lazy Release Consistency for Hardware-Coherent Multiprocessors”. In: *ACM/IEEE Conference on Supercomputing (SC)*. 1995, p. 61. DOI: 10.1145/224170.224398.

[Lam77] L. Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904.

[Lam78] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[Lam79] L. Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[LF00] A.-C. Lai and B. Falsafi. “Selective, accurate, and timely self-invalidation using last-touch prediction”. In: *International Symposium on Computer Architecture (ISCA)*. 2000, pp. 139–148. DOI: 10.1145/339647.339669.

[Lin] Linux Kernel Organization. *Linux Kernel Archive*. URL: http://kernel.org.

[Liu+12] D. Liu, Y. Chen, Q. Guo, T. Chen, L. Li, Q. Dong, and W. Hu. “DLS: Directoryless Shared Last-level Cache”. In: *CoRR* abs/1206.4753 (2012). URL: http://arxiv.org/abs/1206.4753.

[LPM14] D. Lustig, M. Pellauer, and M. Martonosi. “PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014. DOI: 10.1109/MICRO.2014.38.

[LW95] A. R. Lebeck and D. A. Wood. “Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors”. In: *International Symposium on Computer Architecture (ISCA)*. 1995, pp. 48–59. DOI: 10.1145/223982.223995.

[MA14] A. Morrison and Y. Afek. “Fence-free work stealing on bounded TSO processors”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014, pp. 413–426. DOI: 10.1145/2541940.2541987.

[Mad+12] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. “An Axiomatic Memory Model for POWER Multiprocessors”. In: *International Conference on Computer Aided Verification (CAV)*. 2012, pp. 495–512. DOI: 10.1007/978-3-642-31424-7_36.

[Man+15] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. “CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015. DOI: 10.1145/2830772.2830782.

[Mar+05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset”. In: *SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99. DOI: 10.1145/1105734.1105747.

[MB92] S. L. Min and J.-L. Baer. “Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps”. In: *IEEE Trans. Parallel Distrib. Syst.* 3.1 (1992), pp. 25–44. DOI: 10.1109/71.113080.

[McL+15] A. McLaughlin, D. Merrill, M. Garland, and D. A. Bader. “Parallel Methods for Verifying the Consistency of Weakly-Ordered Architectures”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2015. URL: http://users.ece.gatech.edu/~amclaughlin7/PACT15_final.pdf.

[McM01] K. L. McMillan. “Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking”. In: *CHARME*. 2001, pp. 179–195. DOI: 10.1007/3-540-44798-9_17.

[MH05] C. Manovit and S. Hangal. “Efficient algorithms for verifying memory consistency”. In: *SPAA*. 2005, pp. 245–252. DOI: 10.1145/1073970.1074011.

[MHS12] M. M. K. Martin, M. D. Hill, and D. J. Sorin. “Why on-chip cache coherence is here to stay”. In: *Commun. ACM* 55.7 (2012), pp. 78–89. DOI: 10.1145/2209249.2209269.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[Min+08] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. “STAMP: Stanford Transactional Applications for Multi-Processing”. In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2008, pp. 35–46. DOI: 10.1109/IISWC.2008.4636089.

[Moo65] G. E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[MPA05] J. Manson, W. Pugh, and S. V. Adve. “The Java memory model”. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2005, pp. 378–391. DOI: 10.1145/1040305.1040336.

[MQT12] A. Muzahid, S. Qi, and J. Torrellas. “Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2012, pp. 363–375. URL: http://iacoma.cs.uiuc.edu/iacoma-papers/micro12.pdf.

[MS09] A. Meixner and D. J. Sorin. “Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures”. In: *IEEE Trans. Dependable Sec. Comput.* 6.1 (2009), pp. 18–31. DOI: 10.1109/TDSC.2007.70243.

[MS91] K. L. McMillan and J. Schwalbe. “Formal verification of the Gigamax cache consistency protocol”. In: *ISSM International Coherence on Parallel and Distributed Computing*. 1991. URL: http://www.kenmcmil.com/pubs/ISSMM91.pdf.

[Net93] R. H. B. Netzer. “Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs”. In: *Workshop on Parallel and Distributed Debugging*. 1993, pp. 1–11. DOI: 10.1145/174266.174268.

[NN94] S. K. Nandy and R. Narayan. “An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems”. In: *International Workshop on Parallel Processing*. 1994. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.7184&rep=rep1&type=pdf.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. “A Better x86 Memory Model: x86-TSO”. In: *TPHOLs*. 2009, pp. 391–407. DOI: 10.1007/978-3-642-03359-9_27.

[PD00] F. Pong and M. Dubois. “Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models”. In: *IEEE Trans. Parallel Distrib. Syst.* 11.9 (2000), pp. 989–1006. DOI: 10.1109/71.879780.

[PD96] S. Park and D. L. Dill. “Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions”. In: *SPAA*. 1996, pp. 288–296.

[PD97] F. Pong and M. Dubois. “Verification Techniques for Cache Coherence Protocols”. In: *ACM Comput. Surv.* 29.1 (1997), pp. 82–126. DOI: 10.1145/248621.248624.

[PD98] F. Pong and M. Dubois. “Formal Verification of Complex Coherence Protocols Using Symbolic State Models”. In: *J. ACM* 45.4 (1998), pp. 557–587. DOI: 10.1145/285055.285057.

[Pla+98] M. Plakal, D. J. Sorin, A. Condon, and M. D. Hill. “Lamport Clocks: Verifying a Directory Cache-Coherence Protocol”. In: *SPAA*. 1998, pp. 67–76. DOI: 10.1145/277651.277672.

[Plo81] G. D. Plotkin. “A Structural Approach to Operational Semantics”. In: *JLAP*. 1981.

[Pon+98] F. Pong, M. C. Browne, G. Aybay, A. Nowatzyk, and M. Dubois. “Design Verification of the S3.mp Cache-Coherent Shared-Memory System”. In: *IEEE Trans. Computers* 47.1 (1998), pp. 135–140. DOI: 10.1109/12.656100.

[Pug+10] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. “SWEL: hardware cache coherence protocols to map shared data onto shared caches”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010, pp. 465–476. DOI: 10.1145/1854273.1854331.

[Qia+13] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. “Volition: scalable and precise sequential consistency violation detection”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013, pp. 535–548. URL: http://iacoma.cs.uiuc.edu/iacoma-papers/asplos13_3.pdf.

[QM12] X. Qin and P. Mishra. “Automated generation of directed tests for transition coverage in cache coherence protocols”. In: *Design, Automation and Test in Europe (DATE)*. 2012, pp. 3–8. DOI: 10.1109/DATE.2012.6176423.

[RG01] R. Rajwar and J. R. Goodman. “Speculative lock elision: enabling highly concurrent multithreaded execution”. In: *International Symposium on Computer Architecture (ISCA)*. 2001, pp. 294–305. DOI: 10.1109/MICRO.2001.991127.

[RJ15] A. Ros and A. Jimborean. “A Dual-Consistency Cache Coherence Protocol”. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2015. URL: http://ditec.um.es/~aros/papers/pdfs/aros-ipdps15.pdf.

[RK12] A. Ros and S. Kaxiras. “Complexity-effective multicore coherence”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 241–252. DOI: 10.1145/2370816.2370853.

[RK15] A. Ros and S. Kaxiras. “Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting”. In: *International Symposium on Computer Architecture (ISCA)*. 2015. URL: https://www.it.uu.se/katalog/steka984/ISCA15_Alberto_Ros.pdf.

[RLS10] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. “Specifying and dynamically verifying address translation-aware memory consistency”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010, pp. 323–334. DOI: 10.1145/1736020.1736057.

[Roy+06] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. “Fast and Generalized Polynomial Time Memory Consistency Verification”. In: *International Conference on Computer Aided Verification (CAV)*. 2006, pp. 503–516. DOI: 10.1007/11817963_46.

[SA15] H. Sung and S. V. Adve. “DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015, pp. 545–559. DOI: 10.1145/2694344.2694356.

[SAA15] M. D. Sinclair, J. Alsop, and S. V. Adve. “Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015. DOI: 10.1145/2830772.2830821.

[Sah+95] A. Saha, N. Malik, B. O’Krafka, J. Lin, R. Raghavan, and U. Shamsi. “A simulation-based approach to architectural verification of multiprocessor systems”. In: *Computers and Communications*. 1995. DOI: 10.1109/PCCC.1995.472515.

[Sar+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. “Understanding POWER multiprocessors”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2011, pp. 175–186. DOI: 10.1145/1993498.1993520.

[SC97] K. Skadron and D. Clark. “Design issues and tradeoffs for write buffers”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1997. DOI: 10.1109/HPCA.1997.569650.

[Sco13] M. L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[SD87] C. Scheurich and M. Dubois. “Correct Memory Operation of Cache-Based Multiprocessors”. In: *International Symposium on Computer Architecture (ISCA)*. 1987, pp. 234–243. DOI: 10.1145/30350.30377.

[Sha+08] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz. “Verification of chip multiprocessor memory systems using a relaxed scoreboard”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2008, pp. 294–305. DOI: 10.1109/MICRO.2008.4771799.

[Shi+11] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas. *Library Cache Coherence*. Tech. rep. MIT-CSAIL-TR-2011-027. 2011. URL: http://hdl.handle.net/1721.1/62580.

[SHW11] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00346ED1V01Y201104CAC016.

[Sin+13] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. "Cache coherence for GPU architectures". In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 578–590. DOI: 10.1109/HPCA.2013.6522351.

[SK12] D. Sanchez and C. Kozyrakis. "SCD: A scalable coherence directory with flexible sharer set encoding". In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2012, pp. 129–140. DOI: 10.1109/HPCA.2012.6168950.

[SKA13] H. Sung, R. Komuravelli, and S. V. Adve. "DeNovoND: efficient hardware support for disciplined non-determinism". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013, pp. 13–26. DOI: 10.1145/2451116.2451119.

[SP94] M. Srinivas and L. M. Patnaik. "Genetic Algorithms: A Survey". In: *IEEE Computer* 27.6 (1994), pp. 17–26. DOI: 10.1109/2.294849.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. 1992.

[SPA94] SPARC International, Inc. *The SPARC Architecture Manual: Version 9*. 1994.

[Tia+08] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. "Dynamic recognition of synchronization operations for improved data race detection". In: *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2008, pp. 143–154. DOI: 10.1145/1390630.1390649.

[VF96] F. Vavak and T. C. Fogarty. "Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments". In: *International Conference on Evolutionary Computation (ICEC)*. 1996, pp. 192–195.

[VKG14] K. Vora, S. C. Koduru, and R. Gupta. "ASPIRE: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM". In: *ACM SIGPLAN International Conference on Object-Oriented Programming (OOPSLA)*. 2014, pp. 861–878. DOI: 10.1145/2660193.2660227.

[Wal92] D. A. Wallach. “PHD: A Hierarchical Cache Coherent Protocol”. PhD thesis. 1992.

[WB08] I. Wagner and V. Bertacco. “MCjammer: Adaptive Verification for Multi-core Designs”. In: *Design, Automation and Test in Europe (DATE)*. 2008, pp. 670–675. DOI: 10.1109/DATE.2008.4484755.

[WGK90] D. A. Wood, G. A. Gibson, and R. H. Katz. “Verifying a Multiprocessor Cache Controller Using Random Test Generation”. In: *IEEE Design & Test of Computers* 7.4 (1990), pp. 13–25. DOI: 10.1109/54.57906.

[Woo+95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. In: *International Symposium on Computer Architecture (ISCA)*. 1995, pp. 24–36. DOI: 10.1145/223982.223990.

[Xio+10] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. “Ad Hoc Synchronization Considered Harmful”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010, pp. 163–176. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Xiong.pdf.

[YMG96] X. Yuan, R. G. Melhem, and R. Gupta. “A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence”. In: *International Conference on Parallel Processing (ICPP)*. 1996, pp. 114–121.

[Zeb+09] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. “A tagless coherence directory”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 423–434. DOI: 10.1145/1669112.1669166.

[Zha+14] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin. “PVCoherence: Designing Flat Coherence Protocols for Scalable Verification”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2014. DOI: 10.1109/HPCA.2014.6835949.

[ZSD10] H. Zhao, A. Shriraman, and S. Dwarkadas. “SPACE: sharing pattern-based directory coherence for multicore scalability”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010, pp. 135–146. DOI: 10.1145/1854273.1854294.