A Safety-First Approach to Memory Models

by

Abhayendra Narayan Singh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2016

To my family

ACKNOWLEDGEMENTS

I have been very fortunate to have Professor Satish Narayanasamy as my advisor. He is the source of many ideas in this dissertation. He has been a constant source of inspiration and motivation throughout my PhD. I would like to express my most sincere and profound gratitude to him for his support and masterly guidance. I would also like to thank Professor Thomas Wenisch, Professor Peter Chen, Professor Zhengya Zhang and Principal Researcher Madanlal Musuvathi for their support and their role on my committee. I must also thank my undergraduate advisor Professor Mainak Chaudhuri, who introduced me to computer architecture and taught me how to conduct research. Without his encouragement, I doubt if I had pursued PhD in computer architecture.

I also consider myself very lucky for having an very exciting and fruitful collaboration with Dr. Daniel Marino, Professor Todd Millstein, and Madanlal Musuvathi. This collaboration has provided many excellent research and learning opportunities throughout these years. I have benefited immensely from working with them.

Many thanks go to my friends and colleagues in Computer Engineering Laboratory and in Computer Science department for making it a fun place to work. I must thank Jie Yu and Dongyoon Lee for helping me with SIMICS, LLVM and engaging in many useful discussions related to research. During my final year, Chun-hung helped a lot with EECS 483 that I was teaching. Shaizeen collaborated with me during later stages and was fun and helpful. Other friends in Computer Engineering Laboratory I was lucky to interact with include Gaurav Chadha, Ritesh Parikh, Ankit Sethia, Daya Khudia, David Meisner, Joe Greathouse, Shantanu Gupta, Biruk Mammo, Mehrzad Samadi, Aasheesh Kolli, Neha Agarwal and Shruti Padmanabha. I also thank Mukesh Bachhav, Megha Bachhav, Animesh Banerjee, Ayan Das, Vivek Joshi, Soumya Kundu, Mohit Nahata, Gaurav Pandey and many other friends in Ann Arbor who made last six years fun and enjoyable despite having long Michigan winters.

Finally, I must thank my parents and my brother for all they have done to help me reach where I stand now. I sincerely acknowledge the support and encouragement I received from them during my stay at Michigan. I thank all other family members for being loving and caring.

The "DRFx Memory Model" (Chapter III) and "End-to-End Sequential Consistency" (Chapter IV) work were result of collaboration with Dan Marino, Todd Millstein, Madan Musuvathi and Satish Narayanasamy. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

The "Efficiently Enforcing Strong Memory Ordering in GPUS" (Chapter V) was the result of collaboration with Shaizeen Aga and Satish Narayanasamy. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Chapter III contains material that appears in "Efficient Processor Support for

DRFx, a Memory Model with Exceptions", in Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, Madan Musuvathi. The dissertation author was the primary investigator and author of this paper. Portions of Chapter III are Copyright ©2011 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Chapter IV contains material that appears in "End-to-End Sequential Consistency", in 39th Annual International Symposium on Computer Architecture, Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madan Musuvathi. The dissertation author was the primary investigator and author of this paper. Portions of Chapter IV are Copyright ©2012 IEEE. Reprinted, with permission, from [Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madan Musuvathi, End-to-End Sequential Consistency, ISCA, June/2012].

Chapter V contains material that appears in "Efficiently Enforcing Strong Memory Ordering in GPUS", in 48th International Symposium on Microarchitecture, Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. The dissertation author was the primary investigator and author of this paper. Portions of Chapter V are Copyright

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

A Safety-First Approach to Memory Models

by

Abhayendra Narayan Singh

Chair: Satish Narayanasamy

Sequential consistency (SC) is arguably the most intuitive behavior for a shared-memory multithreaded program. It is widely accepted that language-level SC could significantly improve programmability of a multiprocessor system. However, efficiently supporting end-to-end SC remains a challenge as it requires that both compiler and hardware optimizations preserve SC semantics.

Current concurrent languages support a relaxed memory model that requires programmers to explicitly annotate all memory accesses that can participate in a data race ("unsafe" accesses). This requirement allows compiler and hardware to aggressively optimize unannotated accesses, which are assumed to be data-race-free ("safe" accesses), while still preserving SC semantics. However, unannotated data races are easy for programmers to accidentally introduce and are difficult to detect, and in such cases the safety and correctness of programs are significantly compromised.

This dissertation argues instead for a safety-first approach, whereby every memory operation is treated as potentially unsafe by the compiler and hardware unless it is proven otherwise.

The first solution, DRFx memory model allows many common compiler and hardware optimizations (potentially SC-violating) on unsafe accesses and uses a runtime support to detect potential SC violations arising from reordering of unsafe accesses. On detecting a potential SC violation, execution is halted before the safety property is compromised.

The second solution takes a different route and instead of throwing exceptions, it guarantees SC by making both the compiler and the hardware SC-preserving. SC-preserving compiler and hardware are also built on the safety-first approach. All memory accesses are treated as potentially unsafe by the compiler and hardware. SC-preserving compiler and hardware rely on different static and dynamic techniques to identify safe accesses. Our final result indicates that supporting SC at the language level is not very expensive in terms of performance and hardware complexity.

The dissertation also explores an extension of this safety-first approach for data-parallel accelerators such as Graphics Processing Units (GPUs). There are significant micro-architectural differences between a CPU and a GPU, which warrant a fresh look at trade-offs involved in different memory models for GPUs. Furthermore, these differences also render prior efficient SC solutions for CPUs ineffective for data-parallel architectures; requiring an SC-preserving solution specific to data-parallel architectures. The proposed solution based on our SC-preserving approach performs nearly on par with the baseline GPU that implements a data-race-free-0 memory model.

CHAPTER I

Introduction

Parallel programming has become increasingly important over last decade as processor industry transitioned to multi-cores. The "Free Lunch" of software development, enabled by aggressive frequency scaling and Moore's Law, has long been over [108]. As a result, mainstream programmers are expected to write parallel programs in order to extract performance out of current and future parallel platforms.

Shared memory parallel programming is the most common programming model for writing parallel programs. In this model, a program consists of multiple threads that share same address space. All threads share the program's data and communicate with each other via simple loads and stores. Shared memory provides easy to understand abstractions to programmers, but writing a correct parallel program is inherently hard. The programmer has to think about how multiple threads from an application interact with each other. For example, in a sequential program, a read is always guaranteed to return a value written by the last write in program order. For a multithreaded program, however, defining last write for a read operation is not straightforward because multiple threads can be writing to the same location.

A parallel system's interface that defines what values a read can return is known as *memory consistency model* or *memory model* in short. Memory consistency model is a language-level contract that programmer can assume and the system (compiler and hardware) must honor. Thus, foundation of concurrency semantics of a language is based on the memory consistency model supported by that language. Designing a memory model involves a balance between two, often conflicting goals: improving programmer productivity with a memory model that matches programmer's intuition, and maximizing system performance with a weak memory model that enables hardware and compiler optimizations.

Sequential consistency (SC) [69] is arguably the most intuitive memory model discussed in the literature. Under SC, memory accesses of a program appear to have executed in a global order consistent with the per-thread program order. This execution order matches with programmer's intuition of parallel program being an interleaving of its constituent threads. Researchers widely agree that providing SC could simplify concurrency semantics of a language as SC matches with programmer's intuition, but they also believe that SC is an unaffordable luxury [50]. The primary reason for such a belief is the requirement of memory accesses being executed in program order within a thread. This requirement disallows popular compiler optimizations such as common subexpression elimination, loop invariant code motion and store buffer optimization employed in hardware. In quest of high performance, therefore, modern languages and hardware have opted for a weaker memory model that is more permissive in allowing compiler and hardware optimizations that reorder memory accesses.

```
X* x = null;
bool init = false;

// Thread t            // Thread u
A: x = new X();        C: if(init)
B: init = true;        D:   x->f++;
```

(a) Original program.

```
X* x = null;
bool init = false;

// Thread t             // Thread u
B: init = true;
                        C: if(init)
                        D:   x->f++;

A: x = new X();
```

(b) Transformed program.

Figure 1.1: Unintuitive execution behavior due to reordering of memory accesses.

Relaxing memory ordering constraints, however, can result in non-intuitive semantics that are hard to understand and reason about. Consider a simple C++ program shown in Figure 1.1a. In this code snippet, one thread is creating an object and notifying other thread by setting the `init` variable. Here programmer expects statement `D` to correctly dereference the pointer `x`. This expectation is borne out of the fact that we usually take two programming abstractions for granted when analyzing a program snippet: *program order*, which requires that instructions in a thread execute one after the other in the order they appear in the program text; and *shared memory*, which requires that memory behave as a map from addresses to values with each memory operation taking effect immediately. SC matches with programmer's intuition by supporting the program order abstraction.

But contrary to intuition, statement `D` can trigger a null-pointer exception. This is because C++ and other mainstream languages [20, 74] provide weaker semantics known as *data-race-free-0* (DRF-0) [4]. *DRF-0 guarantees SC only if all the data races in a program are explicitly annotated.* Two memory operations conflict if they access the same memory location and at least one of them is a write. A *data race* is defined as a pair of conflicting accesses which are executed by different threads and are not ordered by any synchronization operation. The accesses to `init` in the program in Figure 1.1a form a data race as there is no synchronization between read and write to `init`. Failure to annotate this data race can expose the effects of *sequentially valid* (i.e. correct when considered on a single thread in isolation) compiler and hardware transformations. For instance, in the absence of an annotation on the `init` variable, the compiler and hardware are allowed to freely reorder statements `A` and `B` (Figure 1.1b) since there is no data or control dependence between them. However, this transformation can result in a null-pointer exception at statement `D`.

Such counter-intuitive semantics result in increased complexity of the memory model interface provided to programmers. While such programs are typically considered erroneous, data races are easy for programmers to accidentally introduce and difficult to detect. Having either undefined (e.g. C++) or complex semantics (e.g. Java) for data-races can significantly compromise the *safety*, *correctness*, and *debuggability* of programs. Boehm and Adve has shown how a sequentially valid compiler optimization can cause a racy program to jump to an arbitrary code [20]. Debugging an erroneous program execution is also difficult under DRF-0 model because programmer must always assume that there might have been a data race present. Therefore, it may not be sufficient to reason about program's execution using intuitive SC semantics while debugging.

Providing concurrency support that provides such weak safety property goes against the philosophy of safe languages. A safe language protects its own abstractions [92]. Safe languages provide strong guarantees to programmers for all programs that are allowed to execute, obviating large classes of subtle and dangerous errors and cleanly separating a language's interface to programmers from its implementation details. Modern languages (and programmers) have embraced the compelling programmability benefits of safety despite the additional run-time overhead. Specifically, memory- and type-safe languages such as Java, C#, Python, and JavaScript protect the abstraction of memory as a collection of disjoint entities, each with a well-defined structure and set of operations based on its type.

Unfortunately, C++ and Java memory model are reversing this trend by providing multi-threading support that subverts fundamental programming language abstractions, exposes the complexities of compiler and hardware optimizations to programmers, and makes it easy for programmers to shoot themselves in the foot in ways that are difficult to detect and correct. This is the case not only for "unsafe" languages like C and C++ but also for "safe" languages like Java.

To address above mentioned problems, this dissertation focuses on a safety-first approach for designing memory model. The primary reason for these problems with DRF-0 based memory models is the assumption that all unannotated accesses are *safe by default*; compiler and hardware can freely reorder them. The key idea behind our safety-first approach is to treat unannotated memory accesses as potentially racy ("unsafe"), and relax memory ordering constraints only when they are proven to be data-race-free ("safe"). In this way, SC semantics is guaranteed for all programs whether data-race-free or not. Memory accesses can be identified as safe through programmer provided annotations, static, and dynamic analyses. This dissertation describes three instantiations of this safety-first approach to provide SC guarantees.

DRFx memory model provides uniform fail-stop semantics to all programs (including programs with data-races). DRFx memory model treats all memory accesses as *unsafe by default.* Although DRFx allows common compiler and hardware optimizations on such unsafe accesses, it relies on a runtime support to detect potential SC violations. On detecting a potential SC violation, execution is halted by throwing a *memory model exception.* Therefore, DRFx guarantees that an execution of a parallel program is either SC or it is terminated with a *memory model exception* before SC is violated.

End-to-End sequential consistency: A large fraction of legacy codes contains several intentional benign data-races [85]. Throwing exceptions for all such legacy codes would result in backward incompatibility and may not be an ideal approach. To address this problem, the second solution focuses on providing SC for all programs instead of throwing exceptions. Similar to DRFx, it also treats all unannotated data accesses as *unsafe by default* and restricts potentially SC-violating compiler and hardware optimizations to *safe* accesses only. SC-preserving compiler [76, 77] empirically demonstrates that the performance incentive for relaxing the intuitive SC semantics in compiler is much less than previously assumed. SC-preserving hardware follows *unsafe by default* approach and uses a hybrid classification scheme to identify safe accesses. Together, SC-preserving compiler and hardware guarantee language level SC without incurring significance performance or design complexity overhead.

Efficient SC for data-parallel architectures: The DRFx memory model and End-to-end SC proposals target general purpose CPU multi-core systems. In recent years, data-parallel accelerators, more commonly referred to as Graphics Processing Units (GPUs), have started to play an increasingly important role in parallel computing. Modern GPUs are no longer limited to graphics applications, and are also being used as data parallel accelerators for general purpose programs. Programming models such as CUDA [87] and OpenCL [81] have enabled developers to exploit data-parallelism in general-purpose applications using GPUs (referred as GPGPU applications).

Data-parallel architectures represent a class of parallel systems that is significantly different from CPU based multi-cores due to micro-architectural differences. These differences pertain to virtual memory support, instruction execution, number of concurrent threads per core (SM for GPU), presence of a shared memory in SM and partitioned address space, etc. Due to these differences, a new inquiry into trade-offs involved in supporting various memory models is warranted. Following our *unsafe-by-default* approach, we extend the SC-preserving hardware idea to GPUs and propose an efficient SC design that takes these micro-architectural differences into account and performs on par with baseline DRF-0.

The rest of dissertation is organized as follows. Chapter II presents some background material useful in understanding all subsequent chapters. Chapter III describes the DRFx memory model which provides simple, strong guarantees to a programmer while still allowing most common optimizations. Chapter IV presents a SC hardware design that in conjunction with SC-preserving compiler provides end-to-end SC guarantees to programmers. Chapter V discusses how presents an implementation to support SC efficiently in GPUs. Finally Chapter VI discusses some related work and Chapter VII concludes.

CHAPTER II

Background

This chapter provides background on memory consistency models and data races. It provides useful context for material presented in subsequent chapters.

2.1 Relaxed Memory Models at ISA Level

Early works on memory models delegated most of memory model concerns to the hardware. Therefore, most of the memory models described in literature are hardware centric and defined at assembly language level. SC requires that memory operations from same thread appear to have executed in the program order. However, many sequentially valid optimizations (i.e. correct when considered on individual thread in isolation) can potentially violate SC by reordering memory accesses. In order to allow such optimizations, several memory models have been proposed that relax the program order execution requirement. Tutorial by Adve and Gharachorloo [3] provides a very good overview of these memory models. Total Store Ordering [113] allows later loads to bypass earlier pending stores. Current x86 processors support a variant of

TSO memory model. Weak Ordering [34] relaxes all memory ordering constraints for loads and stores. It classifies all memory operations into *data* and *synchronization* operations. Compiler and hardware are allowed to freely reorder *data* accesses as long as they preserve ordering for accesses to a single location. Release Consistency [42] further divides the *synchronization* operations into *acquire* and *release* operations. It allows roach-motel kind of optimizations, where *data* operations outside of a synchronized region can be brought within the synchronized region. However, memory operations inside a synchronization region can not get outside of the synchronized region. Commercial implementations like Digital Alpha, SPARC V9 RMO, IBM PowerPC, etc. implement either weak ordering, release consistency or some variant of these.

2.2 High Level Language Memory Model

A high level language memory model is consistency model provided by a high level language (such as C++, Java, Python, etc.) to programmers. Ideally a high level language memory model shields programmers from quirks of underlying hardware's memory model. Programmers should be able to reason about a parallel program's execution using rules defined by the language level memory model. A parallel system's observable consistency model depends on the weakest memory model supported in the computing stack. If hardware supports a memory model which is weaker than what a high level language guarantees, either compiler or runtime system has to insert additional *synchronization* operations to prevent the hardware from reordering memory accesses in such a way that violates the language level memory model.

2.2.1 Data-Race-Free-0 (DRF-0) Memory Model

Memory model adopted by current mainstream languages is *data-race-free-0* [4]. It is a variant of weak ordering and is more programmer centric. As in weak ordering, it also classifies all memory accesses into *data* and *synchronization* accesses and guarantees SC execution only for programs which do not contain any data races. Two memory accesses conflict if they access same location and at least one of them is a write. A program contains a data race (or simply a race) if it has a sequentially consistent execution in which two threads are about to execute a conflicting pair of accesses which are not ordered by any synchronization operation. Conflict among synchronization operations is not treated as a data-race. A program is said to be data race free if all conflicting accesses in the program are either properly synchronized or labeled as synchronization operations.

2.2.2 C++ and Java Memory Model

In recent years, there have been significant efforts to bring together language, compiler, and hardware designers to standardize the memory model for mainstream programming languages. The consensus has been around memory models based on DRF-0, which attempts to strike a middle ground between simplicity for programmers and flexibility for compiler and hardware. To make a program data race free, programmers can use synchronization constructs (including high level constructs like lock, barrier, as well as individual memory accesses qualified with `volatile` [74] or `atomic` [20] keyword). Key assumption made by these models is that all *data* accesses are *safe by default*; compiler and hardware can freely reorder *data* accesses as long as they preserve program ordering for accesses to same location. This assumption allows many sequentially valid compiler optimizations that are potentially SC violating (e.g. common subexpression elimination, loop invariant code motion, register promotion, etc.). Similarly hardware optimizations such as out-of-order execution and store buffering, which are also potentially SC violating, are valid in DRF-0 based memory models. While enabling these performance optimizations in compiler and hardware, DRF-0 puts additional burden on programmers to properly annotate all data races in the program as synchronization operations. Failing to annotate even a single data race could result in executions that are non intuitive and could compromise safety of the system as discussed in next section.

2.3 Problems with Relaxed memory models

Under DRF-0, a program containing a data race does not have any defined semantics for its executions. While such programs are typically considered erroneous, data races are easy for programmers to accidentally introduce and are difficult to detect. The DRF-0 memory model, therefore, poses following problems for programmer:

- A racy execution that is allowed to execute arbitrarily can compromise desired safety property. For example, Boehm and Adve have shown how a sequentially valid compiler optimization can cause a program to jump to arbitrary code in presence of a data race [20].

- Debugging an erroneous program is difficult under DRF-0. Programmer must always assume worst case and assume that a data race might be present in the program. Programmer, therefore, can no longer use intuitive SC semantics during debugging to understand and identify errors.

Similar to DRF-0, recently adopted C++ memory model does not provide any defined semantics to programs with data races. On the other hand, Java, being a safe language, can not afford to have undefined semantics. Therefore, it provides a complex semantics that is hard to understand and reason about.

The primary reason for the undefined or complicated semantics in presence of data races is the *safe by default* assumption for *data* accesses made by DRF-0 based memory models. While this assumption enables many common compiler and hardware optimizations, it can also lead to executions that are non-intuitive and could compromise safety. Instead, we need a memory model that preserves safety properties and provides uniform and easy to understand semantics to all programs (even to those with data races). Such a memory model should treat memory accesses as *unsafe by default* as opposed to the *safe by default* assumption of DRF-0. Under *unsafe by default* assumption, compiler or hardware are allowed to reorder memory operations only when they can prove that concerned accesses are data-race-free ("safe"). This approach ensures that safety is not compromised even in presence of data races. Providing uniform semantics to all programs will improve programmability as it preserves *shared memory* and *program order* abstractions that programmers rely on. In Chapters III, IV and V, we discuss novel solutions that follow the *unsafe by default* approach and provide strong SC guarantees to all programs while preserving safety guarantees.

CHAPTER III

DRF*x* Memory Model

Current mainstream languages (C++, Java) support a memory consistency model that is based on *data-race-free-0* (DRF0) memory model. DRF0 memory model provides strong guarantees of SC only to programs that do not have any data races. For programs containing data races, programmer is given either weaker semantics (Java) or no semantics at all (C++). This undermines the ease of debugging and the safety of parallel programs. This is also a problem for compiler/hardware designers as proving the correctness and safety of various compiler and hardware optimizations remains to be a challenge [112, 24].

To address the problem of data races in DRF0 based memory models, researchers have proposed to use data race detectors to identify data races statically or dynamically and provide fail-stop semantics by either rejecting the program (static approach) or halt the execution on detecting a data race (dynamic approach) [5, 19]. However, most static data race detection techniques are limited to lock based synchronization and either restrict programming style or have difficulty in scaling to large programs. On the other hand, dynamic data race detection must be precise, neither allowing a program to complete its execution after a data race nor allowing a race-free execution to be erroneously rejected. However, precise data race detection can slow down the program by 8X when done in software [38]. Hardware based dynamic race detectors have lower overhead in comparison to software based techniques, but they rely on complex rollback and re-execution support in order to avoid false positives [5, 84].

The research described in this chapter presents the DRFx memory model, which provides uniform fail-stop semantics for all programs (including those with data races) without relying on a precise data race detector.

3.1 The DRFx Memory Model

The DRFx memory model provides a simple and strong guarantee to programmers while supporting many common compiler and hardware optimizations. DRFx memory model is inspired by the observation of Gharachorloo and Gibbons [40] that to provide a useful guarantee to programmers, it suffices to detect only the data races that cause SC violations. They illustrate that such detection for a compiled program can be much simpler than full-fledged data-race detection. DRFx follows the *unsafe by default* principle and treats all accesses as being unsafe. While it allows common compiler and hardware optimizations on these unsafe accesses, it uses a runtime system support to detect SC violations arising from reordering of unsafe accesses. On detecting a potential SC violation, DRFx halts the program's execution by throwing a *memory model (MM) exception*. DRFx guarantees two key properties for any program P:

- **Data-Race Completeness**: If an execution is terminated with an MM exception, then P has a data race.

- **SC Soundness**: If an execution is not terminated with an MM exception, then that execution is SC.

Together these two properties imply the DRF0 property: data-race-free programs obtain SC semantics (and are never terminated with an MM exception). However, unlike DRF0, the SC Soundness property allows programmers to safely reason about *all* programs, whether data-race-free or not, using SC semantics. Finally, the Data-Race Completeness property ensures that MM exceptions cannot be raised willy-nilly, but only when the program has a data race.

While these properties provide strong guarantees to programmers, they are carefully designed to admit implementation flexibility. For example, DRFx allows an MM exception to be thrown even if SC is not violated, as long as the original program has a data race. This is an acceptable result since, as with the DRF0 memory model, DRFx considers a data race to be a programmer error. Conversely, DRFx also allows a data-racy execution to continue without exception as long as it does not violate SC. As we will see, our compiler and hardware designs make good use of this flexibility.

SC Soundness requires only that an SC violation will cause execution to halt with an MM exception *eventually*, which also provides implementers significant flexibility. However, an execution's behavior is undefined between the point at which the SC violation occurs and the exception is raised. The DRFx model therefore guarantees an additional property:

- **Safety:** If an execution of P invokes a system call, then the observable program state at that point is reachable through an SC execution of P.

Intuitively the above property ensures that any system call in an execution of P would also be invoked with exactly the same arguments in some SC execution of P. This property ensures an important measure of safety and security for programs by prohibiting undefined behavior from being externally visible.

3.1.1 A Compiler and Hardware Design for DRF*x*

Gharachorloo and Gibbons [40] describe a hardware mechanism to detect SC violations. Their approach dynamically detects *conflicts* between concurrently executing instructions. Two memory operations are said to conflict if they access the same memory location, at least one operation is a write. While simple and efficient, their approach guarantees the SC Soundness and Race Completeness properties with respect to the *compiled* version of a program but does not provide any guarantees with respect to the original *source* program [40, 25].

A key contribution of DRF*x* is the design and implementation of a detection mechanism for SC violations that properly takes into account the effect of both compiler optimizations and hardware reorderings while remaining lightweight and efficient. The approach employs a novel form of cooperation between the compiler and the hardware. DRF*x* introduces the notion of a *region*, which is a single-entry, multiple-exit portion of a program. The compiler partitions a program into regions, and both the compiler and the hardware may only optimize within a region. Each synchronization access must be placed in its own region, thereby preventing reorderings across such accesses. It is also required that each system call be placed in its own region, which allows DRFx to guarantee the Safety property. Otherwise, a compiler may choose regions in any manner in order to aid optimization and/or simplify runtime conflict detection. Within a region, both the compiler and hardware can perform most standard sequentially valid optimizations. For example, unrelated memory operations can be freely reordered within a region, unlike the case for the traditional SC model.

To ensure the DRFx model's SC Soundness and Race Completeness properties with respect to the original program, it suffices to detect *region conflicts* between concurrently executing regions. Two regions R_1 and R_2 conflict if there exists a pair of conflicting operations (o_1, o_2) such that $o_1 \in R_1$ and $o_2 \in R_2$. Such conflicts can be detected using runtime support similar to conflict detection in transactional memory (TM) systems [53]. As in TM systems, both software and hardware conflict detection mechanisms can be considered for supporting DRFx. A hardware detection mechanism is pursued, since the required hardware logic is fairly simple and is similar to existing bounded hardware transactional memory (HTM) implementations such as Sun's Rock processor [33], Intel's Haswell [59], and IBM's Blue Gene/Q [49]. A DRFx compiler can bound the number of memory bytes accessed in each region, enabling the hardware to perform conflict detection using finite resources. While small regions limit the scope of compiler and hardware optimizations, Section 3.5 discusses an approach in that regains most of the lost optimization potential. DRFx hardware leverages several optimizations described in Section 3.5 that allow the hardware to execute and commit regions out-of-order, coalesce regions to reduce the number of conflict checks, and exploit temporal locality to exclude a significant fraction of accesses from participating in conflict detection. These optimizations significantly improve upon the performance overhead of the baseline hardware design for SC violation detection.

3.1.2 Contributions

The research presented in this chapter makes the following contributions:

- The DRFx memory model for concurrent programming languages is defined via three simple and strong guarantees for programmers (Section 3.2). A set of conditions on a compiler and hardware design that are sufficient to satisfy these conditions is established..

- A detailed compiler and micro-architecture design is discussed that instantiates DRFx memory model (Section 3.4 and 3.5). DRFx compliant compiler ensures that regions have a bounded size, allowing a processor to detect conflicts using finite hardware resources. A novel approach to regain most of the lost optimization potential due to small region sizes described. The hardware detects conflicts *lazily*, and several optimizations are described to basic detection mechanism. A DRFx-compliant C compiler is implemented by modifying LLVM [70] and DRFx-compliant hardware designs are modeled using the Simics-based FeS2 simulator [37].

- A detailed evaluation of DRFx-compliant compiler and hardware measures the performance cost in terms of lost optimization opportunity for programs in the Parsec and SPLASH-2 benchmark suites (Section 3.6). The results show that the performance overhead is on average 9.6% when compared to the baseline fully optimized implementation.

3.2 Overview of DRFx

This section provides an overview of DRFx memory model. It first motivates the problem by delving into details of some of issues mentioned in Chapter II, including interaction of data races and relaxed memory models and impracticality of full race detection for the purpose of simplifying the semantics provided to programmers. A description of the DRFx approach for SC violation detection follows.

3.2.1 Compiler Transformations in the Presence of Races

It is well known that *sequentially valid* compiler transformations, which are correct when considered on a single thread in isolation, can change program behavior in the presence of data races [4, 42, 74]. Consider the C++ example from Figure 1.1(a) discussed in Chapter I. Thread `t` uses a Boolean variable `init` to communicate to thread `u` that the object `x` is initialized. Note that although the program has a data race, the program will not incur a null dereference on any SC execution.

Consider a compiler optimization that transforms the program by reordering instructions `A` and `B` in thread `t`. This transformation is sequentially valid, since it reorders independent writes to two different memory locations. However, this reordering introduces a null dereference (and violates SC) in the interleaving shown in Figure 1.1(b).[1] The same problem can occur as a result of out-of-order execution at

[1] Although this "optimization" may seem contrived, many compiler optimizations, for example common-subexpression elimination and loop-invariant code motion, can have the effect of reordering accesses to shared memory.

```
X* x = null;
atomic bool init = false;

// Thread t            // Thread u
A: x = new X();        C: if(init)
B: init = true;        D:   x->f++;
```

Figure 3.1: Correct, data-race-free version of program from Figure 1.1

the hardware level.

To avoid SC violations, languages have adopted memory models based on the DRF0 model [4]. Such models guarantee SC for programs that are free of data races. The data race in this example program can be eliminated by explicitly annotating the variable `init` as `atomic` (`volatile` in Java 5 and later). This annotation tells the compiler and hardware to treat all accesses to a variable as “synchronization”. As such, (many) compiler and hardware reorderings are restricted across these accesses, and concurrent conflicting accesses to such variables do not constitute a data race. As a result, the revised C++ program shown in Figure 3.1 is data-race-free and its accesses cannot be reordered in a manner that violates SC.

3.2.2 Writing Race-Free Programs is Hard

For racy programs, on the other hand, DRF0 models provide much weaker guarantees than SC. For example, the proposed C++ memory model [20] considers data races as errors akin to out-of-bounds array accesses and provides no semantics to racy programs. This approach requires that programmers write race-free programs in order to be able to meaningfully reason about their program’s behavior. But races are a common flaw, and thus it is unacceptable to require a program be free of these

```
X* x = null;
bool init = false;

// Thread t            // Thread u
A: lock(L);            E: lock(M)
B: x = new X();        F: if(init)
C: init = true;        G:   x->f++;
D: unlock(L);          H: unlock(M)
```

Figure 3.2: An incorrect attempt at fixing the program from Figure 1.1.

bugs in order to reason about its behavior. As an example, consider the program in Figure 3.2 in which the programmer attempted to fix the data race in Figure 1.1(a) using locks. Unfortunately, the two threads use different locks, an error that is easy to make, especially in large software systems with multiple developers.

Unlike out-of-bounds array accesses, there is no comprehensive language or library support to avoid data race errors in mainstream programming languages. Further, like other concurrency errors, data races are nondeterministic and can be difficult to trigger during testing. Even if a race is triggered during testing, it can manifest itself as an error in any number of ways, making debugging difficult. Finally, the interaction between data races and compiler/hardware transformation can be counter-intuitive to programmers, who naturally assume SC behavior when reasoning about their code.

3.2.3 Detecting Data Races Is Expensive

This problem with prior data-race-free models has led researchers to propose to detect and terminate executions that exhibit a data race in the program [5, 19, 36]. Note that it is not sufficient to only detect executions that exhibit a strictly simul-

```
// Thread t           // Thread u

A: lock(L);
C: init = true;
                     E: lock(M)
                     F: if(init)
                     G:   x->f++;
                     H: unlock(M);
B: x = new X();
D: unlock(L);
```

(a)

```
// Thread t           // Thread u

A: lock(L);
C: init = true;
B: x = new X();
D: unlock(L);
                     E: lock(M)
                     F: if(init)
                     G:   x->f++;
                     H: unlock(M)
```

(b)

Figure 3.3: A program with a data race may or may not exhibit SC behavior at runtime. (a) Interleaving that exposes the effect of a compiler reordering. (b) Interleaving that does not.

taneous data race. While the existence of such an execution implies the existence of a data race in the program, other executions can also suffer from SC violations. Figure 3.3(a) shows such an execution for the improperly synchronized code in Figure 3.2. When executing under a relaxed memory model, statements B and C can be reordered. The interleaving shown in Figure 3.3(a) suggests an execution where the racing accesses to `init` do not occur simultaneously, but non-SC behavior (null dereference upon executing statement G) can occur. The execution has a *happened-before* data race [68].

Unfortunately, precise dynamic data-race detection either incurs 8x or more performance overhead in software [38] or incurs significant hardware complexity [95, 84]. The cost is due to the need to build a happened-before graph [68] of the program's dynamic memory accesses in order to detect races. A pair of racy accesses can be executed arbitrarily "far" away from each other in the graph. This increases the overhead of software-based detection and requires hardware-based detection to properly handle events like cache evictions, context switches, etc. Imprecise race detectors can avoid some of these problems [101, 75, 21] but cannot guarantee to catch all SC violations, as required by the DRFx memory model.

3.2.4 Detecting SC Violations is Enough

Although implementing DRFx requires detecting all races that may cause non-SC behavior, there are some races that do not violate SC [40]. Thus, full happened-before race detection, while useful for debugging, is overly strong for simply ensuring executions are SC. For example, even though the interleaving in Figure 3.3(b) contains a happened-before data race, the execution does not result in a program error. The hardware guarantees that all the memory accesses issued while holding a lock are completed before the lock is released. Since the `unlock` at D completes before the `lock` at E, the execution is sequentially consistent even though the compiler reordered the instructions B and C. Therefore, the memory model can safely allow this execution to continue. On the other hand, executions like the one in Figure 3.3(a) do in fact violate SC and should be halted with a memory model (MM) exception.

The Venn diagram in Figure 3.4 clarifies this argument (ignore the `RCF` and `RS` sets for now). `SC` represents the set of all executions that are sequentially consistent with respect to a program P. `DRF` is the set of executions that are data-race free. To satisfy the SC Soundness and Data-Race Completeness properties described in Section 3.1, all executions that are not in `SC` must be terminated and all executions in `DRF` must be accepted. However, the model allows flexibility for executions that are not in `DRF` but are in `SC`: it is acceptable to admit such executions since they

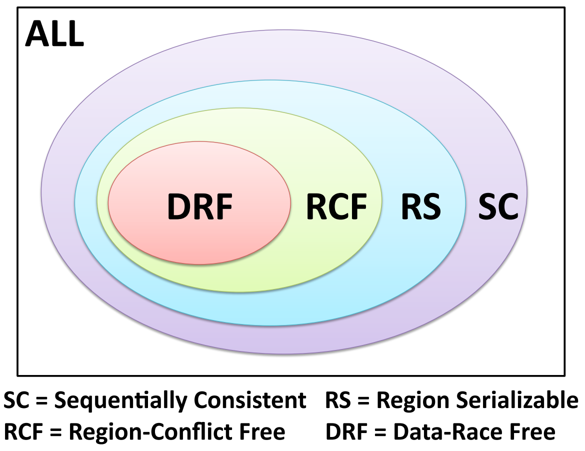


Figure 3.4: The relationships among various properties of a program execution.

are sequentially consistent, but it is also acceptable to terminate such executions since they are racy. This flexibility allows for a much more efficient detector than full-fledged race detection, as described below.

The DRFx memory model only guarantees that non-SC executions *eventually* terminate with an exception. This allows SC detection to be performed *lazily*, thereby further reducing the conflict detector's complexity and overhead. Nevertheless, the Safety property described in Section 3.1 guarantees that an MM exception is thrown before the effects of a non-SC execution can reach any external component via a system call.

3.2.5 Enforcing the DRFx Model

The key idea behind enforcing the DRFx model is to partition a program into regions. Each region is a single-entry, multiple-exit portion of the program. Both the hardware and the compiler agree on the exact definition of these regions and perform program transformations only within a region. Each synchronization operation and each system call is required to be in its own region. For instance, one possible regionization for the program in Figure 3.2 would make each of {B,C} and {F,G} a region and put each lock and unlock operation in its own region.

During execution, the DRFx runtime signals an MM exception if a conflict is detected between regions that are concurrently executing in different processors. We define two regions to conflict if there exists any instruction in one region that conflicts with any instruction in the other region. More precisely, we only need to signal an MM exception if the second of the two conflicting accesses executes before the first region completes. In the interleaving of Figure 3.3(b), no regions execute concurrently and thus the DRFx runtime will not throw an exception, even though the execution contains a data race. On the other hand, in the interleaving shown in Figure 3.3(a), the conflicting regions {B,C} and {F,G} do execute concurrently, so an MM exception will be thrown.

3.2.6 From Region Conflicts to DRFx

The Venn diagram in Figure 3.4 illustrates the intuition for why the compiler and hardware co-design overviewed above satisfies the DRFx properties. If a program execution is data-race-free (DRF), then concurrent regions will never conflict during that execution, i.e., the execution is *region-conflict free* (RCF), so an MM exception will never be raised. Since synchronization operations are in their own regions, this property holds *even in the presence of intra-region compiler and hardware optimizations*, as long as the optimizations do not introduce speculative reads or writes. This reasoning establishes the Data-Race Completeness property of the DRFx model. Further, if an execution is RCF, then it is also *region-serializable* (RS): it is equivalent to an execution in which all regions execute in some global sequential order. That property in turn implies the execution is SC with respect to the original program. Again this property holds even in the presence of non-speculative intra-region optimizations. This reasoning establishes the SC Soundness property of the DRFx model.

In general, each of the sets illustrated in the Venn diagram is distinct: there exists some element in each set that is not in any subset. In some sense this fact implies that the notion of region-conflict detection is *just right* to satisfy the two main DRFx properties. On the one hand, it is possible for a racy program execution to nonetheless be region-conflict free. In that case the execution is guaranteed to be SC, so there is no need to signal an MM exception. This situation was described above for the example in Figure 3.3(b). On the other hand, it is possible for an SC execution to have a concurrent region conflict and therefore trigger an MM exception. Although the execution is SC, it is nonetheless guaranteed to be racy. For example, consider again the program in Figure 3.2. Any execution in which instructions B and C are not reordered will be SC, but with the regionization described earlier some of these executions will trigger an MM exception.

3.2.7 The Compiler and the Hardware Contract

The compiler and hardware are allowed to perform any transformation within a region that is consistent with the single-thread semantics of the region, with one limitation: the set of memory locations read (written) by a region in the original program should be a superset of those read (written) by the compiled version of the region. This constraint ensures that an optimization cannot introduce a data race in

```
for(i=0; i<n; i++)
  sum += a[i];
```

(a)

```
reg = sum;
for(i=0; i<n; i++)
  reg += a[i];
sum = reg;
```

(b)

```
if(n>0) {
  reg = sum;
  for(i=0; i<n; i++)
    reg += a[i];
  sum = reg;
}
```

(c)

Figure 3.5: A transformation that introduces a read and a write.

an originally race-free program.

Many traditional compiler optimizations (constant propagation, common subexpression elimination, dead-code elimination, etc.) satisfy the constraints above and are thus allowed by the DRFx model. Figure 3.5 describes an optimization that is disallowed by the DRFx model. Figure 3.5(a) shows a loop that accumulates the result of some computation in the `sum` variable. A transformation that allocates a register for this variable is shown in Figure 3.5(b). The variable `sum` is read into a register at the beginning of the loop and written back at the end of the loop. However, on code paths in which the loop is never entered, this transformation introduces a spurious read and write of `sum`. While such behavior is harmless for sequential programs, it can introduce a race with another thread modifying `sum`. One way to avoid this behavior is to explicitly check that the loop is executed at least once, as shown in Figure 3.5(c). The DRFx model allows the transformation with this modification, although our current compiler implementation simply disables the transformation. In spite of this, the experimental results in Section 3.6 indicate that the performance reduction due to lost compiler optimizations is reasonable, on average 6.2% on the evaluated benchmarks.

In addition to obeying the requirement above, the hardware is also responsible for detecting conflicts on concurrently executing regions. While performing conflict detection in software would avoid the need for special-purpose hardware, conflict detection in software can lead to unacceptable runtime overhead due to the need for extra computation on each memory access. On the other hand, performing conflict detection in hardware is efficient and lightweight, as demonstrated by the transactional memory (TM) support in several existing processors [33, 59, 49]. DRFx hardware can actually be simpler than TM hardware, since speculation support is not needed. Further, unlike in a TM system, the DRFx compiler can partition a program into regions of bounded size, thereby further reducing hardware complexity by safely allowing conflict detection to be performed with fixed-size hardware resources.

Having the compiler bound the size of regions is essential for efficient hardware detection, but the fences inserted by the compiler for the purposes of bounding should not unnecessarily disallow hardware optimizations. As such, the DRFx implementation supports two types of fences: hard fences that surround synchronization operations and system calls, and soft fences that are inserted only for the purposes of bounding region size. The implementation accounts for the fact that the hardware can perform certain optimizations across soft fences that it must not perform across hard fences.

3.3 Compiler and Hardware Implementation

There are several possible compiler and hardware designs that meet the requirements necessary to ensure the DRFx properties as described in the previous section.

The next two sections describe one concrete approach for DRFx-compliant compiler and hardware. It is evaluated in the section 3.6. The approach is based on two key ideas crucial for a simple hardware design:

- **Bounded regions:** First, the compiler bounds the size of each region in terms of number of memory accesses it can perform dynamically using a conservative static analysis. Bounding ensures that the hardware can perform conflict detection with *fixed-size* data structures. Detecting conflicts with unbounded regions in hardware would require complex mechanisms, such as falling back to software on resource overflow, that are likely to be inefficient.

- **Soft fences:** When splitting regions to guarantee boundedness, the compiler inserts a *soft* fence. Soft fences are distinguished from the fences used to demarcate synchronization operations and system calls which are called *hard* fences. While hard fences are necessary to respect the semantics of synchronization accesses and guarantee the properties of DRFx, soft fences merely convey to the hardware the region boundaries across which the compiler did not optimize. These smaller, soft-fence-delimited regions ensure that the hardware can soundly perform conflict detection with fixed-size resources. But, it is in fact safe for the hardware to reorder instructions across soft fences whenever hardware resources are available, essentially erasing any hardware performance penalty due to the use of bounded-size regions.

3.4 DRFx-compliant Compiler

A DRFx-compliant compiler was built by modifying the LLVM compiler [70]. To ensure the DRFx properties the compiler must simply partition the program into valid regions, optimize only within regions, avoid inserting speculative memory accesses, and insert fences at region boundaries.

3.4.1 Inserting Hard Fences for DRFx-compliance

A hard fence is similar to a traditional fence instruction. The hardware ensures that prior instructions have committed before allowing subsequent instructions to execute and the compiler is disallowed from optimizing across them. To guarantee SC for race-free programs, the compiler must insert a hard fence before and after each synchronization access. On some architectures, the synchronization access itself can be translated to an instruction that has hard-fence semantics (e.g., the atomic `xchg` instruction in AMD64 and Intel64 [20]), obviating the need for additional fence instructions. In the current implementation, the compiler treats all calls to the `pthread` library and lock-prefixed memory operations as "atomic" accesses. In addition, since the LLVM compiler does not support the `atomic` keyword proposed in the new C++ standard, all `volatile` variables are treated as atomic. All other memory operations are treated as data accesses.

To guarantee DRFx's Safety property, a DRFx-compliant compiler should also insert hard fences for each system call invocation, one before entering the kernel mode and another after exiting the kernel mode. Any state that could be read by the system call should first be copied into a thread-local data structure before the first hard fence is executed. This approach ensures that the external system can observe only portions of the execution state that are reachable in some SC execution. Transforming system calls in this way is not implemented in the compiler used for the experiments in Section 3.6.

To insert a hard fence, the compiler uses the `llvm.memory.barrier` intrinsic in LLVM with all ordering restrictions enabled. This ensures that the LLVM compiler passes do not reorder memory operations across the fence. LLVM's code generator translates this instruction to an `mfence` instruction in x86 which restricts hardware optimizations across the fence.

3.4.2 Inserting Soft Fences to Bound Regions

In addition to hard fences, the compiler inserts soft fences to bound the number of memory operations in any region. Soft fences are inserted using a newly created intrinsic instruction in LLVM that is compiled to a special x86 no-op instruction which can be recognized by the DRFx hardware simulator as a soft fence. The compiler employs a simple and conservative static analysis to bound the number of memory operations in a region. While overly small regions do limit the scope of compiler optimizations, experiments show that the performance loss due to this limitation is about 6.2% on average (Section 3.6). After inserting all the hard fences described earlier, the compiler performs function inlining. Soft fences are the inserted in the inlined code. A soft fence is conservatively inserted before each function call and return, and before each loop back-edge. Finally, the compiler inserts additional soft fences in a function body as necessary to bound region sizes. The compiler performs a conservative static analysis to ensure that no region contains more than R memory operations, thereby bounding the number of bytes that can be accessed by any region. The constant R is determined based on the size of hardware buffers provisioned for conflict detection.

The above algorithm prevents compiler optimizations across loop iterations, since a soft fence is inserted at each back-edge. However, it would be possible to apply a transformation similar to loop tiling [115] which would have the effect of placing a soft fence only once every R/L iterations, where L is the maximum number of memory operations in a single loop iteration. Restructuring loops in this way would allow the compiler to safely perform compiler optimizations across each block of R/L iterations.

3.4.3 Compiler Optimization

After region boundaries have been determined, the compiler may perform its optimizations. Any sequentially valid optimization is allowed within a region, as long as it does not introduce any speculative reads or writes since they can cause false conflicts. As such, in the current implementation, all speculative optimizations in LLVM are explicitly disabled.[2] Note, however, that there are several useful speculative optimizations that have simple variants that would be allowed by the DRFx model. For example, instead of inserting a speculative read, the compiler could insert a special

[2] The LLVM implementation has functions called `isSafeToSpeculativelyExecute`, `isSafeToLoadUnconditionally` and `isSafeToMove`, which were modified to return `false` for both loads and stores.

prefetch instruction which the hardware would not track for purposes of conflict detection. The Itanium ISA has support for such speculation [111] in order to hide the memory latency of reads. Also, as shown earlier in Figure 3.5, loop-invariant code motion is allowed by the DRFx model, as long as the hoisted reads and writes are guarded to ensure that the loop body will be executed at least once.

3.5 DRFx-compliant Hardware: Design and Implementation

This section discusses the proposed DRFx processor architecture. A lazy conflict detection scheme using bloom filter signatures is described, as well as several optimizations that allow efficient execution in spite of the small, bounded regions created by the DRFx compiler.

3.5.1 Overview

To satisfy DRFx properties, the runtime has to detect a conflict when region-serializability may be violated due to a data race and raise a memory model exception (Section 3.2.6). Figure 3.8 presents an overview of a DRFx hardware design which supports this conflict detection. Additions to the baseline DRF0 hardware are shaded in gray. The state of several hardware structures at some instant of time during an execution of a sample program is also shown. Section 3.5.11 discusses the implementation details of the proposed design.

Rollback is a necessary requirement of hardware transactional memory systems. As such, they can easily tolerate false positives in their conflict detection mechanism by simply rolling back and re-executing. This allows them to use cache-line granularity conflict detection which may report false races. DRFx, on the other hand, does not require a rollback mechanism. But, because it terminates an execution upon detecting a race, false race reports cannot be tolerated. As such, DRFx performs byte-level conflict detection. Performing precise, *eager* byte-level conflict detection complicates the coherence protocol and cache architecture [73]. For instance, such a scheme would require the hardware to maintain byte-level access state for every cache block, maintain the access state even after a cache block migrates from one processor to another, and clear the access state in remote processors when a region commits.

Instead, the DRFx hardware employs lazy conflict detection [47]. Each processor core has a *region buffer* which stores the physical addresses of memory accesses executed in a region. An entry is created in the region buffer when a memory access is committed from the reorder buffer (ROB). A load *completes* its execution when it *commits* from the ROB, while a store *completes* its execution when it *retires* from the store buffer. When all the memory accesses in a region have completed their execution, the processor broadcasts the address set for the region to other processors for conflict checks. Once the requesting processor has received acknowledgments from all other processors indicating a lack of conflicts, it commits the region and reclaims the region buffer entries. The communication and conflict check overhead is reduced by using bloom filter based signatures to represent sets of addresses [27]. A *signature buffer* is employed to store the read and write signatures for all the in-flight regions in a processor core.

The region buffer has to be at least as large as the maximum number of instructions allowed to be executed in a soft-fenced region created by the DRFx compiler. The static analysis used by the DRFx compiler to guarantee this bound is necessarily conservative and may create regions that are much smaller than the desired bound. Frequent soft-fences leads to frequent conflict checks. This cost is reduced by coalescing adjacent regions separated by a soft fence into a single region at runtime when there is sufficient space available in the region buffer. Supporting this optimization requires using a region buffer somewhat larger than the maximum possible region-size guaranteed by the compiler.

When executing a hard fence, the DRFx hardware stalls the execution of all future memory accesses until all accesses preceding the fence have completed. This helps guarantee correct behavior of synchronization operations and ensures that any conflicts that are detected indeed correspond to a data race. But it also prevents full utilization of processor resources since instruction and memory level parallelism cannot be exploited across the fence. If the more frequently occurring soft fences behaved the same as hard fences, these lost opportunities to exploit parallelism would result in significant performance overhead. Fortunately, this is unnecessary since soft fences do not indicate the presence of synchronization. In fact, memory accesses from a region can be allowed to execute even if earlier regions that end in soft fences have not committed. In addition, regions separated by a soft fence can be committed out of order.

3.5.2 Signature-based Lazy Conflict Detection

Let us assume that a processor treats soft fences similar to hard fences, an assumption that will be relaxed later in the discussion. DRFx hardware employs lazy conflict detection to detect when region-serializability could have been violated due to a data-race.

Each processor core has a *region buffer* to record the addresses of memory locations accessed in a region. Each region buffer entry corresponds to a cache block and stores cache block's address along with read and write bit vectors that keep track of the bytes accessed within a cache block. This organization of region buffer entry allows us to coalesce memory accesses from same region if they access the same cache block. The DRFx compiler bounds the size of a soft-fenced region to a predefined bound `B`, which determines the minimum size that a processor needs to provision for a region buffer. In practice, however, a region would require fewer entries in the region buffer as memory accesses to same cache block are coalesced.

Similar to DRF0 hardware, the memory accesses within a region can execute out-of-order, and in the case of stores, retire from a store buffer out-of-order. An entry in the region buffer is created for a memory access when it is committed from the ROB.

Once all the memory accesses of a region have committed from the ROB, and stores are retired from the store buffer, the corresponding processor broadcasts the address set to the other processors to perform conflict checks. On receiving a conflict check request, a processor detects a conflict if the addresses in its region buffer intersect with the address set received. If the intersection is empty, an acknowledgment is sent to the requester. On receiving acknowledgments from all the other processors, a processor commits a region by deleting its address entries from the region buffer.

Broadcasting addresses accessed by every region and checking their membership in every processor's region buffer is clearly expensive. To reduce this cost, bloom filter based signatures [27] can be used. Memory addresses accessed by a region are represented using a read and a write signature. Signatures for the in-flight regions are stored in the *signature buffer* (more than one region could be in-flight due to the out-of-order execution optimizations discussed later in Section 3.5.5). To perform conflict checks for a region, a processor first broadcasts only its signatures. Each processor performs AND operations over the incoming signatures with the contents in its signature buffer. On detecting a potential conflict, a `NACK` is sent to the requester. On receiving a `NACK`, a processor sends the full address set for the region so that precise conflict detection can be performed.

The size of signatures needs to be large enough so that false conflicts are rare, avoiding frequent transmission of full address sets. On the other hand, large signatures could incur significant communication overhead. But, since many regions can be in flight in a processor at once, the signature may be compared with many remote regions, increasing the probability of getting a false conflict. To address this problem, large signatures (1024 bits) are used, but they are compressed before transmission to reduce communication overhead. To build these signatures, cache block address of locations accessed by memory operations are used. Using cache block granularity keeps the number of unique addresses that go into signatures low. Because many regions have small access sets, their signatures are effectively compressed using a simple, efficient run-length encoding scheme. This strategy resulted in very high compression ratios which significantly reduced communication overhead.

3.5.3 Concurrent Region Conflict Check and Region Execution

When a processor `P` receives a conflict check request for a region `R′`, it need not stall the execution of its current region `R` while it performs the conflict check. A conflict check can be performed in parallel with the execution of a local region. The intuition here is that any memory access in pipeline can be shown to have executed after memory accesses in `R′`. Thus, hardware can order `R′` before `R` in the region serialization of the execution.

However, care must be taken in ensuring that memory accesses in pipeline are completed after memory accesses in `R′`. Stores are not completed before they are committed from store buffer. A store in pipeline, therefore, is definitely not completed. If an in-flight load has already read the value from the cache, DRFx hardware needs to ensure that value has not been changed by another region that is serialized before current region. To this end, a load that has already read the data from cache, is re-executed if the accessed cache block is invalidated/evicted before load is committed from the ROB. This mechanism is same as the speculative load-execution technique proposed by Gharachorloo *et al.* [41] for a SC hardware. A DRF0 hardware could use this mechanism to enable load speculation across fences. However, in the proposed DRFx design, this mechanism is enforced for all loads (even if they are not speculating across fences). Re-executing *all* loads on cache invalidations is certainly expensive, but such re-executions are expected to be rare.

3.5.4 Coalescing Soft-Fence-Bounded Regions

The DRF*x* compiler uses a conservative static analysis to estimate the maximum number of instructions executed in a region. This could result in frequent soft fences. But a processor can dynamically ignore a soft fence if the preceding soft-fenced region executed fewer memory accesses than a predetermined threshold `T`. Combining two contiguous soft-fenced regions at runtime does not violate DRF*x* guarantees, because any conflict detected over the newly constructed larger region is possible only if there is a race, and ensuring serializability of the larger, coalesced soft-fenced regions is sufficient to guarantee SC for the original unoptimized program.

However, the processor needs to ensure that the newly constructed region does not exceed the size of its region buffer. The design guarantees this by using a region buffer that is of size `T + B`, where `B` is the compiler specified bound for a soft-fenced region, and `T` is the threshold used by a processor to determine when to ignore a soft fence. Too high a value for the threshold `T` would result in large regions at runtime, which might negatively impact performance, because the probability of aliases in signatures increase. Also, it could undermine out-of-order commit optimization discussed in Section 3.5.6.

3.5.5 Out-of-Order Execution of Regions

In this optimization, the restriction on execution of soft-fenced regions is relaxed and soft-fenced regions are allowed to be executed out-of-order. In the case of a hard fence, before a processor can execute memory accesses from a later region, it has to wait for all memory accesses in the preceding regions to complete. This is clearly a requirement for hard fences, since false conflicts may be detected if memory accesses are allowed to be reordered across hard fences that demarcate synchronization operations. However, this execution ordering can be relaxed for soft fences, allowing multiple regions that are not committed to be in-flight simultaneously. For example, in Figure 3.6, $\texttt{I}_7$ can be allowed to execute even if regions $\texttt{R}_0$ and $\texttt{R}_1$ have pending memory accesses in the ROB or the store buffer. If there is a pending store in a previous region (e.g., $\texttt{I}_1$), its value can be forwarded to a load in a later region (e.g., $\texttt{I}_7$).

The correctness of the above optimization can be intuitively understood by observing that executing memory accesses out-of-order only results in more in-flight accesses that could potentially conflict. Therefore, it does not mask any conflicts that would have been detected before. Also, reordering accesses across soft fences will not cause any access to be reordered across a synchronization operation. As such, any conflict that is detected as a result of this reordering still implies the presence of a data race.

3.5.6 Out-of-order Commit of Regions

Once a region's memory accesses have been completed, a processor can initiate conflict check and commit the region from the region buffer if the check succeeds. Since instructions are committed from the ROB in the program order, it is guaranteed that when a region is ready to commit, all the memory accesses from preceding regions would have also committed from the ROB. There could, however, be stores in the store buffer pending for the earlier regions. As a result, those earlier regions would not yet

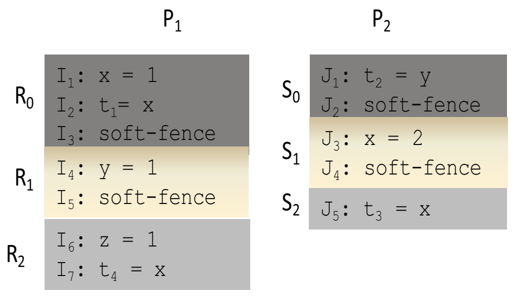


Figure 3.6: An Example Binary Compiled Using DRF*x* Compiler.

be ready to commit. In this scenario, it is correct to conflict check and commit a later region as long as all its accesses have committed from the ROB and retired from the store buffer. When a region is being committed out-of-order while there are pending regions before it, it must also include the addresses of memory locations accessed by earlier regions during conflict detection. This inclusion is required to ensure that there are no conflicts with earlier regions when a region is being committed out-of-order.

For example, in Figure 3.6, say region R_0 is waiting for its store I_1 to be retired from the store buffer. In the meantime, I_4 has completed and has retired from the store buffer. Now R_1 is ready to commit. The processor can perform conflict checking for R_1 (including the addresses for any uncommitted, prior regions), and if no conflict is detected, commit by deleting its entries from the region and signature buffers (but leaving the entries for uncommitted, prior regions). This optimization can be intuitively understood by observing that even if a write from R_0 lingering in the store buffer eventually causes a conflict with another access that has not been committed from the ROB yet, the successful conflict check of the addresses in R_1 and R_0 at the time R_1 commits establishes a global order of all committed and lagging regions in the system at that point. This guarantees SC behavior up to the latest committed region in each thread.

3.5.7 Exploiting Locality in Memory Accesses

The hardware design discussed so far conflict checks all the addresses accessed within a region. We propose an optimization that exploits the temporal locality exhibited by applications to drastically reduce the number of addresses that need to be conflict checked for a region.

The key insight is that once a processor core has conflicted checked an address for a read or a write access, it does not need to perform the same check again till it relinquishes the coherence permission that it must have acquired for performing that access.

However, since processor core performs conflict checks at the byte granularity (to ensure precision that is necessary in DRFx memory model), it is possible that a core has coherence read/write permission to an entire cache block, but core has not performed conflict checks for all the addresses in that cache block.

To solve this problem, processor keeps track of two additional *safe* bits per cache block:

- `read-safe`: If set, it indicates that all the addresses of the cache block has been conflict checked for read-permission.
- `write-safe`: If set, it indicates that all the addresses of the cache block has been conflict checked for read and write permission.

If a memory instruction results in a cache miss and brings in a cache block, or if it accesses a cache block for which the corresponding safe bit is not set, it needs to participate in the conflict detection. During conflict detection, read and write signatures are built for only those memory instructions that did not observe appropriate *safe* bit in the cache. Read and write signatures (which are based on cache block address) are sent to other cores in first phase of the conflict detection. If no conflict is detected in signatures, appropriate `read/write-safe` bit is set for the cache blocks if they have not been invalidated yet. However, if a conflict is detected in signatures, core sends the precise read/write address set to perform precise conflict detection. Also, *safe* bits in cache blocks accessed in this region remain unchanged if a conflict in signatures is detected.

When a region is ready to commit from the region buffer, the processor core needs to know the subset of addresses for which it needs to initiate a conflict check. To determine this set, each region buffer entry is extended with `read-safe` and `write-safe` bits. When a read/write accesses a cache block with the read/write safe bit set, then its region buffer entry's safe bit is also appropriately set. The addresses of these accesses are not conflict checked when the region commits, except for the following scenario.

It is possible for a processor core to relinquish coherence permission to a cache block when there is a safe region buffer entry to one or more of its addresses. Therefore, before evicting or downgrading coherence permission to a cache block, the processor core needs to snoop the region buffer and reset the safe bits that correspond to the evicted/downgraded cache block.

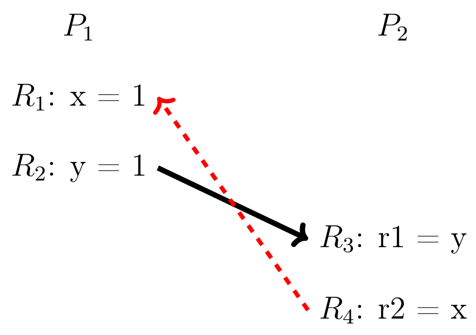


Figure 3.7: Out-of-order commit and locality optimizations together can violate DRFx guarantees

We observe that this locality optimization cannot be employed together with the out-of-order commit optimization discussed in Section 3.5.6, as it can violate the DRFx guarantees. We illustrate this problem using the example shown in Figure 3.7. In this example, two processors (P_1 and P_2) are executing four soft-fenced regions each containing only one instruction. Assume that variables x and y are allocated on separate cache lines C_1 and C_2 respectively and P_2 is caching C_1 with `read-safe` bit set. Also assume that write of x in R_1 misses in the cache and store is pending in the store buffer. In meantime, R_2 completes and starts its commit out of order. Since there are no conflicting accesses in P_2, R_2 successfully commits. Subsequently P_2 executes R_3, R_4 and commits them in program order without detecting any conflict. Note that for R_4, P_2 does not send out address of x to P_1 because P_2 thinks that reading x is safe (indicated by read-safe bit in C_1). Finally, R_1 completes and broadcast address of x, but does not detect any conflict as P_2's region buffer is empty at this time. Therefore, P_1 commits R_1 without raising any MM exception. However, this execution is not SC because execution of regions is not serializable. Therefore, out-of-order commit and cache based locality optimizations are not compatible with each other in preserving

DRFx guarantees.

3.5.8 Handling Context Switches

A thread can incur a context switch at runtime for a variety of reasons. When possible, DRFx require that the context switch be delayed until the subsequent soft fence instruction. As regions are bounded in the number of memory instructions, most well-behaved programs will eventually execute a soft fence after a finite amount of time. To account for adversarial programs that can perform unbounded computation (while still performing a bounded number of memory accesses), DRFx requires that a DRFx-compliant compiler inserts additional soft fences in regions that could potentially execute unbounded number of instructions. By doing so, it is possible to delay scheduler-induced thread context switches without affecting the fairness of the operating system scheduler. For such delayed context switches, the hardware waits until all prior regions are committed and performs the context switch when the region buffer is empty.

Certain context switches, such as those induced by page faults and device interrupts, are critical and cannot be delayed. DRFx-style conflict detection should be disabled for low-level system operations such as the page-fault handler. It is unclear if halting such critical functionality with a memory-model exception is a good design choice. Instead, such low-level code be (either manually or statically) verified to be data-race free. This observation leads to a simple solution that does not require virtualizing the region buffer.

When critical context switches occur, the processor retains the region buffer entries for the switched out thread. When the processor is executing the page-fault or the interrupt handler, it continues to perform conflict detection on behalf of the switched-out thread. Since conflict detection is disabled for the handler, no new entries are added to the region buffer. When the handler has finished, the operating system is required to schedule the switched-out thread on the same processor core to avoid false detection. At this point, the thread continues using the region buffer, which contains the same entries it had at the time it was switched out.

While a page fault is being serviced for a thread, a processor can execute other threads instead of waiting for the data to be fetched from the disk. Processor core can can allow `N` context switches while handling a page-fault by provisioning a region buffer with a size that is `N` times that of the maximum bound specified by the compiler. For example, if the compiler bounds the region size to 64 locations and region buffer size is 512, DRFx hardware can allow 8 context switches.

3.5.9 Debugging Support

When a program is terminated with an MM exception, the processor provides the addresses of the starting and ending fence instructions of each conflicting region to assist in debugging.

A processor may encounter non-MM exceptions such as a null-pointer dereference, division by zero, etc., while the current region is yet to complete. In this scenario, the processor stalls the execution of the current region and performs conflict detection for the partially executed region. If the conflict check succeeds, indicating no data race, it raises the non-MM exception. But if a conflict is detected, the processor throws an

MM exception instead.

An MM exception in proposed design is imprecise in the sense that the state of the program when an exception is raised may not be SC. Because, the compiler or hardware could have already performed SC-violating optimizations in regions that contain racing accesses. Even an eager conflict detection scheme can only guarantee that the program state at the time of an exception is SC with respect to the compiled, binary program [73]. The state could still be non-SC with respect to the source program due to compiler optimizations.

3.5.10 System Calls and Safety

The DRFx compiler places each system call in its own region, separated from other regions by hard fences. Furthermore, the compiler generates code to ensure that system calls only access thread-local storage. Any user data potentially read by a system call is copied to thread-local storage before executing the preceding hard fence, and any user data written by the system call is copied out of thread-local storage after the succeeding hard fence.

An adversarial program may not obey the DRFx compiler requirement that every region's size be bounded to a predefined limit. When a program executes a region that exceeds the bound, the DRFx hardware can trivially detect that condition and raise an MM exception to ensure safety.

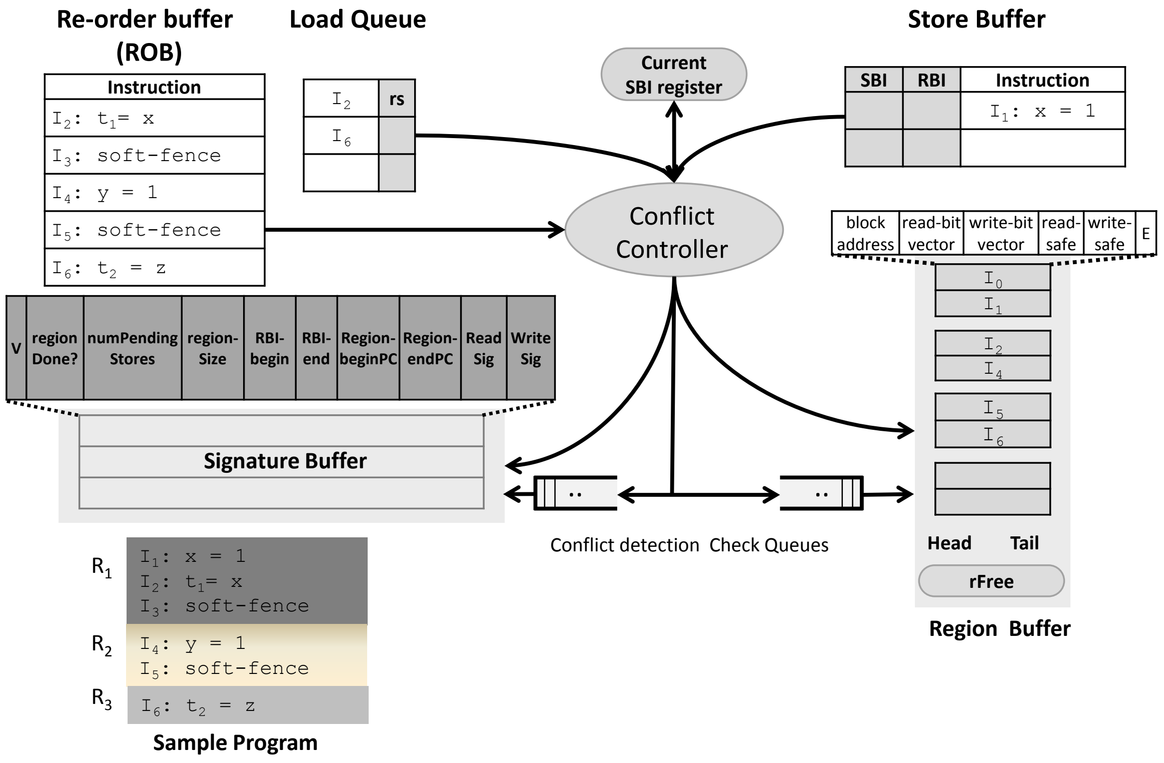


Figure 3.8: Architecture support for DRF*x* (shown in gray).

3.5.11 DRF*x* Hardware Design Details

Region and signature buffers for each processor core are the main extensions to the baseline hardware structures. This design assumes a snoop-based architecture which is extended with additional messages to support conflict checking. Conflict check messages are independent of coherence messages. Figure 3.8 shows the DRF*x* hardware extensions to a baseline out-of-order processor with store buffer. In the proposed design, regions are committed in-order, and locality in memory accesses is exploited as discussed in Section 3.5.7. A detailed description of these extension follows.

When a hard fence is committed from the ROB, a new region is created by first finding a free entry in the signature buffer (one with its `valid` bit unset), initializing the entry's fields, and storing its index in the current Signature Buffer Index (SBI) register. The `SBI` register keeps track of the signature buffer entry corresponding to the region that is currently being executed. When a soft fence is committed from the ROB, a new region is created only if the current region size (stored in the `region-Size` field of the region's signature buffer entry) exceeds a predetermined threshold `T` (32 in this design). If a hard or soft fence starts a new region, its instruction address is stored in the `Region-beginPC` field of the new region's signature buffer entry. This information is used while reporting an MM exception.

When a memory instruction commits from the ROB, it searches the region buffer to check if there is an entry for same cache block address already present for current region. If no such entry exists, a region buffer entry is allocated for it. The register `rFree` keeps track of the total number of free entries in the region buffer. If no free region entry is available, the memory access is stalled at the commit stage of the pipeline. Free region buffer entries are organized as a free-list. The `head` and `tail` registers point to the first and last entries in the free list respectively. The head and tail of a region is stored in the `RBI-begin` and `RBI-end` fields in the signature buffer entry of that region (RBI stands for Region Buffer Index).

A region buffer entry contains following fields: cache block address, read and write bit vectors to keep track of bytes accessed within the cache block, `read-safe` and `write-safe` bits, and `E` bit that is set when corresponding cache block has been evicted from the cache. When a memory instruction commits from the ROB, it updates appropriate bits in the region buffer entry. If entry is a newly allocated, `read-safe` or `write-safe` are set according to the `read/write-safe` bits observed by the instruction. The `read/write-safe` bits are reset when either corresponding cache block is evicted or a memory instruction with `read/write-safe` bit not set is inserted into this region buffer entry. When a region commits without detecting a conflict in signatures, it sets *safe* bits in the cache only if `E` bit in the region buffer entry is not set.

When a load reads from the cache, it needs to remember whether the accessed cache block had either of `read-safe` or `write-safe` bits set or not. To this end, load queue entries are extended to have additional bit: `rs`, which is set if either of `read-safe` or `write-safe` bits were set in the cache block. On cache block invalidations, load queue is searched and `rs` bit is reset. When load is committed from ROB, it uses `rs` bit to update its region buffer entry. Load's cache block address is inserted into read signature if rs bit is not set. Stores update `write-safe` bit when they retire from the store buffer. To update the region buffer entry directly, region buffer index (RBI) of the corresponding region buffer entry is saved in the store buffer. Similar to loads, they update write signature if `write-safe` bit is not set in the cache.

In order to determine when all stores from a regions have completed, a counter `numPendingStores` is kept in region's signature buffer entry. This counter is incremented when an store is put into the store buffer and is decremented when a pending store retires from the store buffer.

When a hard or a soft fence commits from the ROB, its region's `regionDone` bit is set in the signature buffer. Also, its region's `Region-endPC` is updated with its instruction address. A region is ready to commit, if its `regionDone` bit is set and `numPendingStores` is zero. Before committing a region, its addresses need to be conflict checked with all the in-flight regions in remote processor cores. During this process, the region's SBI is used as its identifier in the conflict check messages.

If the conflict check succeeds, the region is committed by deallocating its entries in the signature and region buffers. The signature buffer entry is identified using the region's SBI used during its conflict check. The start and end of a region's entries in the region buffer are determined using the `RBI-begin` and `RBI-end` fields stored in that region's signature buffer entry. Committed region's region buffer entries are added to the free list.

3.6 Performance Evaluation

This section presents some performance results comparing the performance of programs compiled and executed under the DRFx memory model to those compiled and executed under a DRF0 model.

3.6.1 Methodology

The baseline compiler is the LLVM [70] compiler with all optimizations enabled (similar to compiling with the `-O3` flag in `gcc`) and with fences inserted before and after each call to a synchronization function and each access to a volatile variable.[3] The DRFx compiler is the implementation described in the Section 3.4: hard fences are inserted before each call to a synchronization function and each access to a volatile

[3]The unmodified LLVM compiler using its x86 backend targets hardware obeying the TSO memory model. The baseline simulated architecture uses a weaker memory model which permits additional reorderings not allowed by TSO. As such, compiler inserts the additional fences around synchronization accesses to ensure that the program behaves correctly on the weaker model. The benchmarks run slightly faster in this baseline, DRF0 configuration than on a simulated TSO architecture running code compiled with unmodified LLVM.

Table 3.1: Processor Configuration

Processor	4-core CMP. Each core operating at 2Ghz.
Fetch/Exec/ Commit width	4 instructions (maximum 2 loads or 1 store) per cycle in each core.
Store Buffer	TSO: 64 entry FIFO buffer with 8 byte granularity. DRF0, DRFx: 8 entry unordered coalescing buffer with 64 byte granularity.
L1 Cache	64 KB per-core (private), 4-way set associative, 64B block size, 2-cycle hit latency, write-back.
L2 Cache	1MB private, 4-way set associative, 64B block size, 10-cycle hit latency.
Coherence	MOESI snoop protocol
Interconnection	Hierarchical switch, fan-out degree 4, 512-bit link width, 2-cycle link latency.
Memory	80-cycle DRAM lookup latency.
Region buffer	544 entry, 8 banks, 2-cycle CAM access.
Bloom filter	1024 bits. 2 banks indexed by 9 bit field after address permutation[27]. 2-cycle access latency.

variable, optimizations that perform speculative reads or writes are disabled, and soft fences are inserted to conservatively bound region size to 512 memory accesses.

Both the baseline and DRFx architectures are simulated using a cycle-accurate, execution driven, Simics based x86_64 simulator called FeS2 [37]. The baseline architecture is a 4-core chip multiprocessor operating at 2GHz (Table 3.1). It allows both loads and stores to execute out of order between fences. The DRFx architecture adds support for soft fences and conflict detection as described in the previous section, using a region buffer of size 512 (compiler bound) + 32 (to support region coalescing).

Performance is measured over a subset of the PARSEC [15] and SPLASH-2 [117]

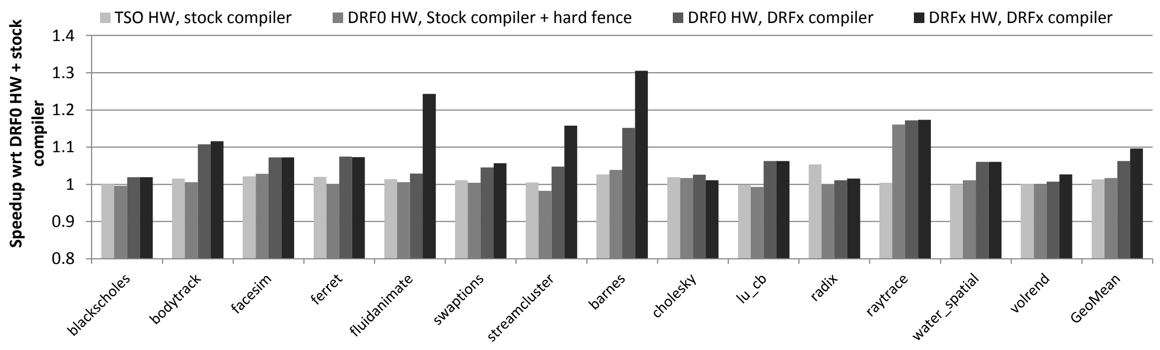


Figure 3.9: Slowdown of benchmark programs run under the DRFx model compared to a baseline DRF0 model, broken down in terms of cost of lost compiler optimization and cost of hardware race detection.

benchmarks. All of these benchmarks are run to completion. For PARSEC benchmarks (blackscholes, bodytrack, facesim, ferret, fluidanimate, streamcluster, swaptions), the `simmedium` input set was used. For SPLASH-2 applications (barnes, cholesky, lu (contiguous), radix, raytrace, water-spatial, and volerand) the default inputs were used.

3.6.2 Comparison of DRFx with Other Relaxed Memory Models

Figure 3.9 compares the performance of TSO, DRF0 and DRFx memory models. The results are normalized to the execution time of DRF0 hardware executing a binary produced by stock LLVM compiler. Since stock compiler is for x86 (TSO memory model), fences were added before and after synchronization operations to ensure correct memory ordering on DRF0 hardware. For DRFx, we measured both the cost of lost compiler optimizations and the cost of conflict detection in hardware. To measure the cost of lost compiler optimizations, we executed the binaries produced by DRFx-compliant compiler on DRF0 hardware that treats soft-fences as no-op. To measure the cost of conflict detection, we evaluate a processor configura-

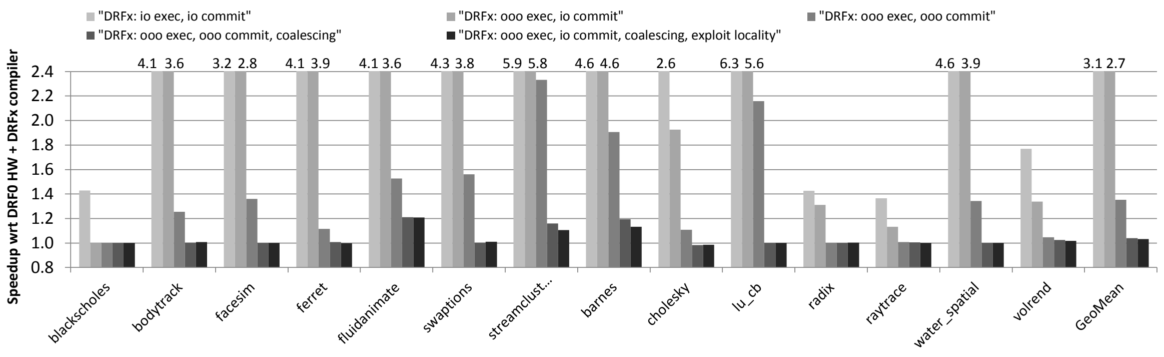


Figure 3.10: Effectiveness of Region Coalescing, and Out-Of-Order Region Execution and Commit Optimizations.

tion that employs optimizations discussed in Section 3.5. In Figure 3.9, we see that a DRFx-compliant compiler (labeled as "DRF0 HW, DRFx Compiler") incurs about 6.2% overhead on average due to restricted compiler optimizations. Conflict detection in hardware adds about 3.4% overhead. In following sections we'll show how optimizations proposed in Section 3.5 are crucial for low conflict detection overhead.

3.6.3 Effectiveness of DRFx Hardware Optimizations

Figure 3.10 demonstrates the importance of distinguishing soft fences and implementing the optimizations described in the Section 3.5. Here performance is measured as execution time normalized to that of DRF0 hardware. When soft fences are treated like hard fences (label "DRFx: io exec, io commit"), the benchmarks experience slowdown of more than 3× on average. Enabling out-of-order execution and commit for soft-fence-bounded region significantly reduces this overhead to about 35.3% on average. Coalescing soft-fenced regions further reduces this overhead to 4.0%. Coalescing of soft-fenced regions is highly effective in increasing the average region size. Average soft-fenced region size without coalescing is about only 8 memory operations.

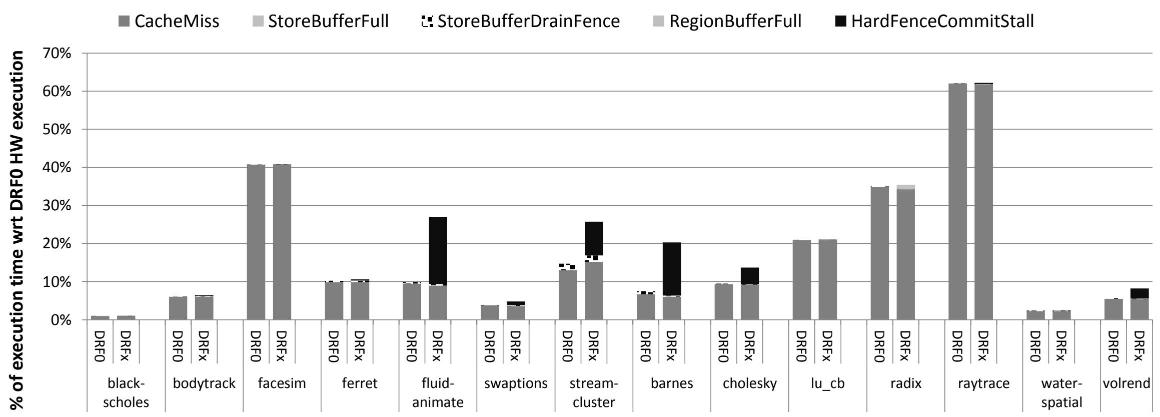


Figure 3.11: Profile of commit stage stall.

Enabling coalescing for soft-fenced region increases their average size to about 1200 memory operations. Increased region size results in lowering the frequency of conflict detection and thus lowering the cost of conflict detection. Enabling optimization for exploiting locality in memory accesses (Section 3.5.7) results in overall best design and has about 3.1% performance overhead. Note that as mention earlier in Section 3.5.7, out-of-order commit is not compatible with optimization for exploiting locality of memory accesses.

To understand the sources of overhead involved in conflict detection, we profile cycles during which commit stage is stalled (Figure 3.11). Since we insert entries into region buffer at commit stage, processor's commit could be stalled if region buffer is full and can not accept any new entry. Such stall is labeled as "RegionBufferFull" in Figure 3.11. Hard fences in DRFx can also introduce additional stall while waiting for all preceding regions (including region ended by the hard-fence) to complete their conflict detection. This stall is labeled as `HardFenceCommitStall`. For some applications, however, conflict detection increases stall cycles up to 17% of the total execution time of DRF0 hardware. High overhead for `fluidanimate` is due to its

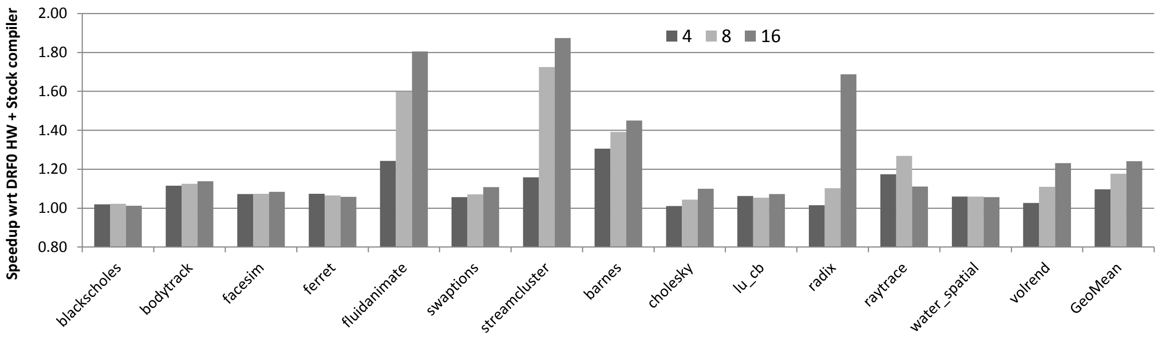


Figure 3.12: Scalability of DRFx with increasing number of cores.

usage of a lot of locks (and hence hard-fences). We also see that finite size of the region buffer has very limited impact on overall execution time (`RegionBufferFull` being almost negligible) primarily due to coalescing of soft-fenced regions.

3.6.4 Scalability

Figure 3.12 shows the scalability of DRFx over 4, 8, and 16 cores. With increasing number of cores, broadcast latency and probability of false conflicts in signatures increases, which in turn results in higher cost of conflict detection. For `fluidanimate`, `barnes`, and `radix` increase in conflict detection overhead with increasing number of cores is attributed mostly to the increase in rate of false conflicts in signatures. On other hand, for `streamcluster`, region size decreases with increasing number of cores. This reduction is attributed to very frequent hard-fences in the program. Smaller regions result in more hard-fences waiting for conflict detection more frequently. This increases the cost of conflict detection as we increase the number of processors. For remaining benchmarks, we observe small increase in conflict detection overhead.

3.7 Conclusion

The DRFx memory model for concurrent programming languages gives programmers simple, strong guarantees for all programs. Like prior data-race-free memory models, DRFx guarantees that all executions of a race-free program will be sequentially consistent. However, while data-race-free models typically give weak or no guarantees for racy programs, DRFx guarantees that the execution of a racy program will also be sequentially consistent as long as a memory model exception is not thrown. In this way, DRFx guarantees safety and enables programmers to easily reason about *all* programs using the intuitive SC semantics. Furthermore, the minor restrictions DRFx places on compiler optimizations are straightforward, allowing compiler writers to easily establish the correctness of their optimizations.

DRFx capitalizes on the fact that sequentially-valid compiler optimizations preserve SC as long as they do not interact with concurrent accesses on other threads. Since performing precise data race detection is impractically slow in software and complex in hardware, DRFx allows the compiler to specify code regions in which optimizations were performed. The hardware can then efficiently target data race detection only at regions of code that execute concurrently. This allows DRFx-compliant compiler and hardware to cooperate, terminating executions of racy programs that may violate SC. The formal development establishes a set of requirements for the compiler and the hardware that are sufficient to obey the DRFx model. The implementation and evaluation indicate that a high-performance implementation of DRFx is possible.

CHAPTER IV

SC-preserving Hardware

Previous chapter described how SC can be preserved in the compiler efficiently. However, to support SC at language level, binaries produced by an SC-preserving compiler must be executed on a SC hardware. This chapter focuses on building an efficient SC-preserving hardware. In past, several efficient SC hardware designs have been proposed, but most of them rely on fairly complex hardware support. The research described in this chapter presents an efficient SC-hardware design that has lower complexity overhead in comparison to earlier proposed SC designs. The key is to treat all accesses as potentially “unsafe” and use static and dynamic techniques to identify “safe” accesses. Relaxing memory ordering constraints for safe accesses does not require complex hardware support. Along with SC-preserving compiler, SC-hardware provides SC guarantees for all programs and addresses the safety and debuggability problem discussed in Chapter II.

4.1 Introduction

Past work has produced efficient SC hardware designs [7, 41, 97, 55, 43, 26, 114, 17] by introducing novel techniques for speculatively reordering memory accesses and recovering when there is a possible SC violation. In-window speculation [41] is relatively simple as it only reorders memory instructions before they are are committed from the reorder buffer (ROB). Commercial processors already implement this optimization to efficiently support x86's Total Store Order (TSO) consistency model [60]. However, in-window speculation alone is insufficient to attain high performance in SC hardware, as loads still cannot be committed until the store buffer is drained. To reap the benefits of a store buffer in SC hardware, researchers have proposed a more aggressive out-of-window speculation technique that reorders even committed memory instructions [97, 43, 26, 17, 114]. But out-of-window speculation and the accompanying recovery mechanisms are arguably quite complex and have not yet been realized in any practical processor implementation.

This chapter presents an SC hardware design that is more complexity-effective than past out-of-window speculation techniques, but still results in an efficient design. It leverages the simple observation that memory model constraints need not be enforced for private locations and shared read-only locations [103, 79, 2]. Since most memory accesses are to private or read-only data [48, 30], this observation provides an opportunity to design an efficient SC hardware by simply relaxing the ordering constraints on many memory accesses, obviating the need for complex speculation techniques.

A modern TSO processor design (which already supports in-window speculation [41]) is extended to exploit the above idea to support SC efficiently. Store buffer is divided into two structures: one is the regular FIFO store buffer that orders stores to shared locations, and the other is a private, unordered store buffer to fast-track stores to private locations. This design allows private and shared, read-only loads to commit from the ROB without a store buffer drain. It also allows a load to a shared read-write location to commit from the ROB without waiting for the private store buffer to drain. Therefore, when compared to the TSO design implemented in today's processors, the only additional memory ordering restriction that proposed SC design imposes is that loads to shared read-write locations are stalled until the FIFO store buffer containing shared stores is drained.

Two complementary techniques are discussed to enable a processor to identify private and shared read-only accesses. The first technique is based on static compiler analysis. An SC-preserving compiler (discussed in previous chapter) is extended to conservatively identify all memory accesses to function locals whose references do not escape their functions. These memory accesses are guaranteed to be private to a thread. The compiler communicates this information to the processor by setting a bit in a memory instruction's machine code.

The compiler analysis necessarily needs to be conservative in classifying a memory access as private. A complementary dynamic technique is employed that extends the hardware memory management unit and operating system's page tables to keep track of private and shared, read-only pages. During address translation, a processor determines the type of a memory access and decides whether or not to enforce memory model constraints for that access. Past work employed a similar dynamic technique to track private pages, but used it to optimize cache performance [48, 66] and directory-based coherence [30] rather than to reduce the overhead due to memory model constraints.

Experimental study on the PARSEC [15], SPLASH [117] and Apache benchmarks shows that the overhead of proposed SC hardware over TSO is less than 5.0% on average. It also shows that the overhead of providing end-to-end SC (running the SC-preserving compiler's output on our SC hardware) when compared to running the stock LLVM compiler's output on a TSO hardware is 7.5% on average.

Although this chapter focuses on designing an efficient SC hardware, the observation that memory model constraints need not be enforced for private and shared, read-only accesses could be similarly exploited to improve the performance of any memory model implementation.

4.2 Efficient *and* Complexity-Effective SC Hardware Remains a Challenge

Prior to discussing the challenges of designing SC hardware, few commonly used terms that are also used in this chapter are clarified. In a modern out-of-order processor, instructions can *execute* out-of-order but must *commit* from the reorder buffer in program order. If allowed by the memory model, a store may commit from the reorder buffer and be placed in a store buffer before its value has been written to cache or memory. The stored value is made *visible* to other threads only when a store *retires* from the store buffer, which is when its value is written to the appropriate memory location in the cache. Two memory accesses in different threads are said to *conflict* if they access the same memory location and at least one of them is a write.

SC hardware needs to guarantee that the memory accesses of a program appear to have executed in a global sequential order that is consistent with the per-thread program order. A naïve SC hardware design would force loads and stores to be executed and committed in program order. Also, a store's value would need to be made visible to all threads atomically when committed. This naïve design disallows most hardware optimizations including out-of-order execution and store buffers.

Even x86 processors' TSO memory model disallows loads from executing out-of-order. As mentioned earlier in previous chapter, modern x86 processor implementations support in-window speculation [41] to reduce the overhead due to this load-load memory ordering constraint of TSO [60]. Loads are allowed to be speculatively executed out-of-order. The processor still commits them in-order and recovers when a possible memory ordering violation is detected between the execution and commit of a load. A violation is detected when a processor core receives a cache coherence invalidation request for a location accessed by a load that has already executed but has not yet committed. The logic that supports recovery from branch misprediction is mostly sufficient to recover from in-window memory ordering violations as well.

The primary performance overhead in TSO, when compared to weaker relaxed consistency models [4], is the cost of enforcing store-store ordering. TSO requires a global total order for all stores, which is guaranteed by committing stores to a FIFO store buffer and retiring them to memory atomically in the program order. As a result, a processor core may have to stall the commit of a store from ROB if the store buffer is full. However, this overhead tends to be small for most programs.

In-window speculation is also useful for optimizing SC hardware since it allows many loads to execute out of order, eliminating much of the overhead in ensuring SC. However, unlike TSO which permits loads to be reordered before stores, SC can not take full advantage of store-buffer optimization. While SC hardware can commit a store from the ROB and place it in the store buffer, any following load cannot be committed from the ROB until the store buffer is drained. That is, all preceding stores need to be retired and their values made visible to other threads before a later load can commit. In-window speculation does not help reduce this costly overhead in an SC hardware design.

Past research has proposed aggressive speculation techniques to allow store-buffer optimization in SC hardware [97, 44, 43, 46, 26, 114, 17]. These designs extend the idea of in-window speculation to speculatively commit loads from the ROB even when the store buffer is not empty. This requires fairly complex hardware that keeps track of the register and memory state before each committed load, detects potential SC violations by comparing incoming coherence invalidation requests with the addresses of committed loads, and performs a rollback when a potential SC violation is detected. To avoid speculation, Lin et al. [72] proposed to check if there is any conflict with pending accesses in remote cores before committing a memory instruction from the ROB. While this design eliminates the need for out-of-window checkpoint and rollback support, it still requires significant changes to the coherence protocol to efficiently perform conflict detection before committing a memory instruction from the ROB.

This chapter proposes an alternative mechanism to reap the benefits store-buffer optimization for a certain class of memory accesses while preserving SC.

4.3 Relaxing Memory Model Constraints for Safe Accesses

Processors enforce memory ordering constraints in order to prevent other processors from being able to observe reordering not allowed by the memory model. Past SC hardware designs have uniformly enforced memory model constraints on all non-atomic memory accesses, distinguishing only between stores and loads. This is overly conservative and unnecessary for a significant fraction of memory accesses.

If either the compiler or the runtime system can guarantee that there can be no conflicting memory access on another thread which could observe or alter the result of a particular memory access, then the processor can safely reorder that access in any manner that preserves intra-thread data dependencies. We refer to memory accesses with this property as *safe* accesses and the rest as *unsafe* accesses.

For instance, if a memory access is to a location that is *private* to the current thread, then clearly there can be no conflicting memory accesses, so the access is safe. A compiler can guarantee this property for all dynamic instances of a static memory instruction that accesses only thread-local data. A runtime system can guarantee this property for any access to a location that it knows has only been accessed by the current thread so far during execution. Once a memory location is accessed by a second thread, the runtime system must detect this situation and require that this and future accesses obey memory model constraints on the processor. A similar idea can be used to identify *shared read-only* memory locations accessed by multiple threads as safe.

The above observation is exploited to design an efficient and complexity-effective SC hardware. Proposed SC hardware design can be understood in relation to out-of-window speculation techniques proposed in the past for reducing the overhead of enforcing memory model constraints. The key insight of those past techniques was to speculatively relax memory ordering restrictions on memory accesses as they are rarely violated. Unfortunately, the required support for recovery is costly in terms of processor complexity. In contrast, memory ordering restrictions are relaxed only for those memory accesses which are guaranteed to be safe by the compiler or runtime system. Since this relaxation is always correct, hardware support for misspeculation recovery is no longer needed, hence resulting in a low-complexity solution.

Over 45% of memory accesses are found to be safe for benchmarks studied (Section 4.8.1). Relaxing of SC memory ordering restrictions is focused on these accesses. But this approach is generally applicable to any memory model. For example, TSO requires that stores be retired in program order from the store buffer, but that restriction need not be enforced for safe stores.

4.4 Design: Memory Access Type Driven SC Hardware

This section discusses a low-complexity, efficient, SC hardware design based on exploiting memory access type information. Figure 4.1 shows the extensions we propose to a baseline TSO processor and operating system used today. Before delving into SC hardware design, we briefly describe the two techniques we use to determine safe memory accesses and how that information is communicated to the hardware.

To simplify the discussion, we assume that a memory instruction accesses only one location in memory. Section 4.5.3 discusses how memory instructions in a CISC architecture that can read or write to multiple locations are handled.

4.4.1 Two Techniques to Determine Memory Access Type

The proposed processor design relies on two complementary techniques to determine safe accesses: a static compiler analysis and a dynamic analysis based on the page protection mechanism.

The static analysis determines memory instructions in a program that are guaranteed to access private or read-only locations (safe locations). It does this using a conservative, intra-procedural analysis to identify function-local variables that do not escape the scope of their functions (safe variables). At run-time, the memory locations of such variables will be private to the thread that invokes the function, so all accesses to these variables are considered safe. [1] Our analysis also considers accesses to constant literals as safe. The Instruction Set Architecture (ISA) is extended to allow a compiler to communicate to the hardware which accesses it guarantees are safe. When a processor core decodes a memory instruction and allocates an ROB entry, it sets a bit (`ss`) in the ROB (Figure 4.1) if that instruction is flagged as safe by the compiler, which is later used to relax memory model constraints. This static approach incurs little runtime complexity, but it has to be conservative and may clas-

[1] Care must be taken to ensure correctness as two function-local variables in different functions may be allocated to the same stack location (Section 4.5).

sify accesses to locations (especially those on the heap) that are actually private as unsafe.

A dynamic technique is also employed that leverages operating system (OS) support for classifying accesses at the page granularity [48]. The OS protects pages at the process-level, which is extended to support thread-level page protection by adding a few fields to the page table entry (Figure 4.1). The first read and/or the first write from a thread will trigger an exception to the OS, which allows the OS to keep track of the state of the page (private, shared read-only, or shared read-write). The TLB entry for a page is also extended with an additional safe bit, which is used to determine if it is a safe page or not. During address translation for a memory access in the execution stage, a processor core determines if the access is to a safe page, and sets the `ds` bit in the ROB, which is later used to relax memory model constraints. Care must be taken to preserve memory ordering constraints between memory accesses when the state of the page changes (Section 4.6).

Even if a page contains only one shared read-write byte, accesses to any part of the page will be treated as unsafe by the dynamic scheme described above. Thus, a static analysis that classifies locations at finer granularity complements the dynamic analysis. In the proposed design, a *hybrid* scheme is used. Since both static and dynamic classification schemes are conservative, it is correct for the hybrid scheme to consider a memory access to be safe if either one of the two methods classifies that access as safe (i.e. either `ss` or `ds` is set in the ROB entry).

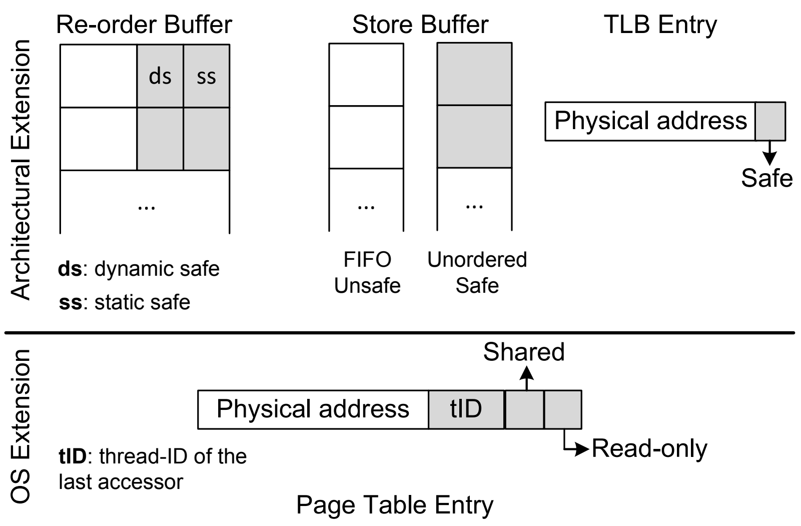


Figure 4.1: Memory Access Type Driven SC Processor and OS

4.4.2 SC Architecture Design

As pointed out in Section 4.2, TSO allows loads to be reordered before stores, which enables store buffer optimization. SC, however, disallows this optimization, which is the only performance cost of SC hardware when compared to TSO hardware (assuming in-window speculation [41] for both designs).

A simple extension can reduce this cost significantly: divide the store buffer into two parts as shown in Figure 4.1. One part is the traditional FIFO store buffer for handling unsafe stores. The second is an unordered store buffer for fast-tracking safe stores. A processor core can determine whether a load/store is safe or not by examining the `ss` and `ds` bits in its ROB entry. This design has the following three main performance advantages when compared to the baseline SC design.

- A safe load can commit from the ROB even when there are pending stores in either or both of the two store buffers (perhaps waiting for their cache misses to be serviced). Thus, proposed design provides TSO performance for safe loads.

- An unsafe load can commit from the ROB even when there are pending stores in the unordered store buffer containing safe stores. Thus, if a safe store is waiting for a cache miss, following unsafe loads need not wait to commit.
- Stores in the unordered store buffer can be coalesced if they access the same cache line. Also, they can be retired out of order. As a result, a safe store need not wait for a pending (safe or unsafe) store to retire. This decreases pressure on store buffer capacity. This property could also be exploited to improve a TSO hardware's performance.

4.4.3 Store-to-Load Forwarding with Two Store Buffers

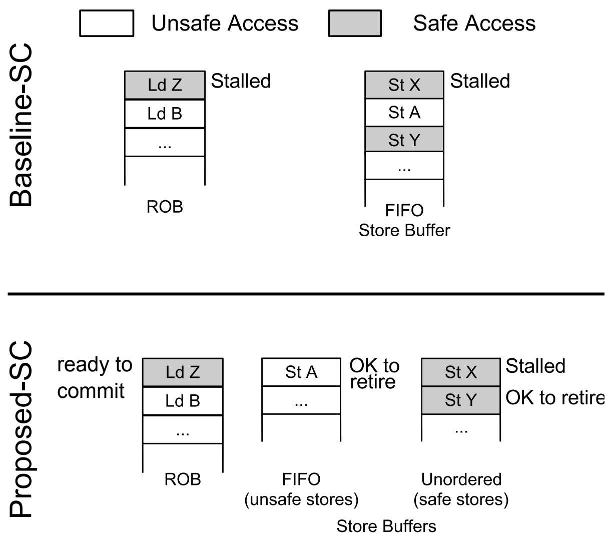
Having two store buffers could potentially complicate store-to-load forwarding logic. This complication is avoided by ensuring that all bytes accessed by a memory instruction are of the same type (safe or unsafe). This property is referred as the *memory-type* guarantee. Furthermore, compiler ensures that for any valid read-after-write dependency the two memory accesses are of the same type. Therefore, to detect store-to-load forwarding for a safe load, only the unordered store buffer needs to be searched. Similarly, an unsafe load needs to search FIFO store buffer only.

The static analysis ensures that all the variables accessed by a memory instruction are of the same type as follows. If any memory instruction could access both safe and unsafe variables, then the static analysis conservatively marks that instruction as unsafe. In addition, any safe variable accessed by that instruction is reclassified as unsafe, as are all other instructions that access those reclassified variables. The read-after-write dependency guarantee is ensured since the compiler uniquely classifies each variable as either safe or unsafe, so both stores and loads to the variable will use the same access type. A discussion of an interesting corner case arising from distinct logical variables being mapped to the same physical address can be found in Section 4.5.2.

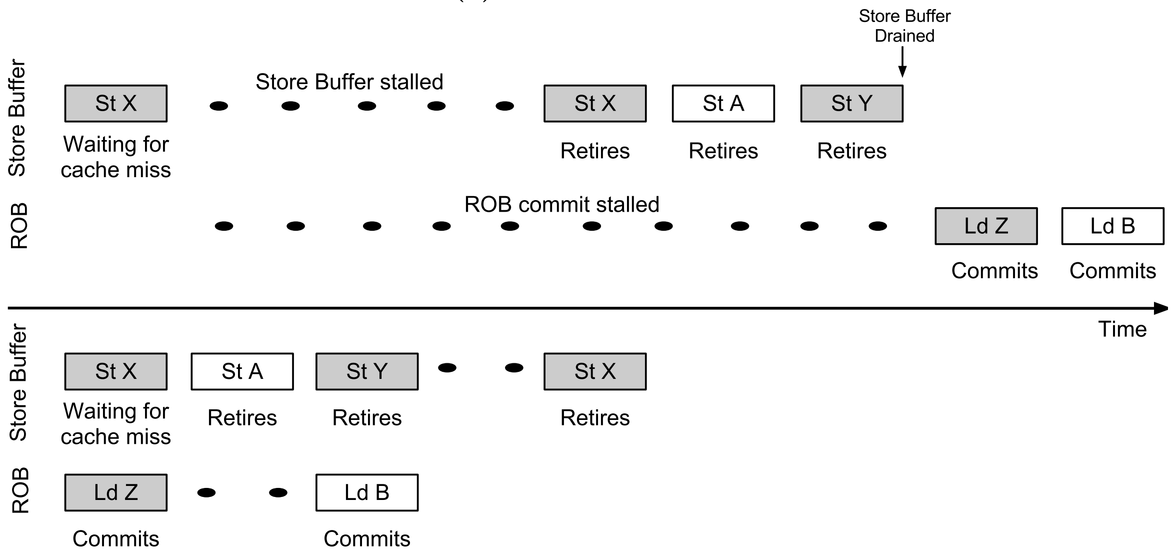
The dynamic analysis could violate the memory-type guarantee only when an instruction accesses memory locations that span multiple pages. Fortunately, current architectures produce multiple micro-operations to execute such unaligned load accesses [61]. As a result, memory-type guarantee is preserved for each load/store micro-operation, which is sufficient to ensure correct store-to-load forwarding. The read-after-write dependency guarantee could only be violated when a page transitions from private or shared-read-only to shared-read-write. But such a transition entails flushing the store buffers (see Section 4.6), thus the guarantee is maintained.

4.4.4 Illustration

Figure 4.2 depicts an example to illustrate the performance advantages of the proposed SC hardware design. The top half of the picture illustrates a baseline SC hardware design and the bottom half illustrates the workings of proposed design. Figure 4.2a represents the initial states of the ROB and the store buffers for a program, and Figure 4.2b shows the events that take place in the store buffer and in the ROB along a timeline. Shaded cells represent safe accesses. Assume that only `St(X)` incurred a cache miss and the rest are cache hits. Finally, for simplicity, assume that the cache has one read and one write port.



(a) Processor States



(b) Execution timeline

Figure 4.2: Comparison of a program's execution in baseline SC (top) and proposed SC hardware (bottom) designs.

The figure shows that, in the baseline design, `St(X)` is safe but is stalled at the head of the store buffer. This unnecessarily stalls the retirement of the following stores and also prevents the loads in the ROB from being committed. The loads in the ROB must wait to commit until after the cache miss is resolved and the store buffer is drained.

In proposed SC design (bottom half of the picture), the long latency `St(X)` is sent to the unordered store buffer. This allows all the following safe (`St(Y)`) and unsafe (`St(A)`) stores to retire. Also, it allows safe (`Ld(Z)`) and unsafe (`Ld(B)`) loads to commit from the reorder buffer. Finally, observe that safe load `Ld(Z)` is allowed to commit even before the preceding unsafe store `St(A)` retires. The only memory ordering enforced is that unsafe load `Ld(B)` must wait to commit until the unsafe store `St(A)` retires, which results in a one cycle stall for the ROB commit. In contrast, in the SC baseline, ROB commit is stalled until the FIFO store buffer becomes empty. This stall can be significant depending on the number of pending stores that miss in the cache and the cache miss latency.

4.4.5 SC Memory Model Guarantees

The SC memory model requires that any program state that is made "externally" visible is *SC-reachable* in the sense that the state is reachable through an SC execution of the source program. We consider the program state read by a synchronous system call and the final program state to be externally visible. By construction, SC-preserving hardware and compiler guarantee that the final program state is SC-reachable. To guarantee that any program state visible to a system call handler is

SC-reachable, hardware only needs to ensure that the store buffers of the processor core invoking the system call are drained before the system call handler is executed. This is already the case even with conventional processor designs that support precise context switches.

However, at an asynchronous interrupt (e.g., interrupt from an interactive debugger), hardware can only guarantee that the program state is SC-reachable for the shared variables but not for the private variables. For the private variables, hardware can only guarantee SC with respect to the compiled binary, because accesses to private variables may have been optimized and reordered by the SC-preserving compiler. But guaranteeing that the program state at an asynchronous interrupt is precise with respect to the source program is a more general problem that is known to be an issue even for sequential programs in the presence of compiler optimizations [1].

4.5 Static Classification of Memory Accesses

This section describes a static approach to classify memory instructions as either safe or unsafe. The compiler communicates this information to the hardware through dedicated bits in a memory instruction's machine code.

In the dynamic scheme described in Section 4.6, implementation efficiency requires that access patterns are tracked at the granularity of a memory page. This means that if a single byte on a page is accessed by multiple threads, then *all* locations on that page must be treated as shared and suffer the performance consequences of strict ordering requirements when accessed. The static scheme described in this section has no runtime detection cost, and as such, nothing prevents us from treating an access to one byte as safe while treating an adjacent byte on the same page as unsafe.

4.5.1 Classification of Memory Accesses

Proposed static analysis runs at compile time and conservatively determines memory accesses that could potentially access mutable shared variables and marks them as unsafe. The remaining accesses to private and read-only shared variables are marked as safe.

The analysis first classifies program variables:

- **Safe variable:** A variable is classified as safe only if the compiler can statically guarantee that it is either a read-only variable or will be accessed by only a single thread during its lifetime.

- **Unsafe variable:** A variable that is not safe is classified as unsafe. It may be accessed by multiple threads during its lifetime.

Once program variables have been classified, the analysis can classify memory accesses:

- **Safe access:** A memory instruction that accesses one or more safe variables and does not access any unsafe variables is classified as safe.

- **Unsafe access:** A memory instruction that accesses one or more unsafe variables and does not access any safe variables is classified as unsafe.

It is possible that a memory instruction accesses both safe and unsafe variables (e.g., an instruction dereferencing a pointer that can map to variables of both types).

We refer to such instructions as "mixed accesses". In order to ensure correct store-to-load forwarding on proposed hardware, all accesses to a variable must be either safe or unsafe. To accomplish this, compiler marks a mixed-access as unsafe *and* also demotes any mutable safe variable that it accesses to an unsafe variable. This step may now cause some safe accesses to become mixed or unsafe accesses. We iterate this step till all accesses are either classified as safe or unsafe.

Sophisticated sharing and thread escape analysis [32, 100] could be used to perform the initial classification of program variables. But rather than use a heavyweight, inter-procedural analysis, the static analysis relies on simple modular information to conservatively determine if an access is safe. Non-constant global variables, static variables, and dynamically allocated heap objects are all considered unsafe. This leaves only function parameters, function locals, and constants as potentially safe variables.

Proposed compiler analysis is built on top of LLVM which already performs a simple analysis to identify non-escaping, function-local variables (i.e. those variables whose address is not taken using the `&` operator). Modified compiler takes advantage of this existing analysis and marks these non-escaping variables as safe. Stack locations used by the compiler for register spilling are also classified as safe. Finally, literals (shared or private) are classified as safe as well.

4.5.2 Ensuring Correctness for Hardware with Two Store Buffers

As mentioned in Section 4.4.3, store-to-load forwarding is only performed between loads and stores of the same memory access type. For instance, an unsafe store to

```
void f(){
  int x;
  x=1;
  printf("%d\n",x);
}
```

(a)

```
void g(){
  int y;
  y=2;
  external_fn(&y);
  printf("%d\n",y);
}
```

(b)

```
void u(){
   union {
     int i;
     char * p;
   };
   p = (char *) ''SC'';
   printf(``%d\n'',i);
   external_fn(&i);
}
```

(c)

Figure 4.3: Local variables (`x`, `y` and `z`) in different functions in a program may map to the same physical location, complicating the unsafe versus safe distinction. Union members (`i` and `p` in function `u()`) must have same memory access type.

a memory location `L` which is queued in the unsafe FIFO store buffer will not be forwarded to a safe load from `L`. Thus, maintaining correct program semantics requires that the compiler mark a load and a store that access the same memory location with the same access type.

The algorithm described above maintains this invariant for accesses to a location within a function: only non-escaping local variables and compiler temporaries, neither of which can have aliases, are marked as safe. Furthermore, demoting safe variables touched by mixed accesses to unsafe and reclassifying the variables' accesses guarantees that all accesses are either entirely to safe or unsafe variables.

However, this intra-procedural analysis does not account for location reuse across different functions. Consider the example functions shown in Figures 4.3a and 4.3b. Both function `f` and `g` contain a single local variable. In `f`, compiler marks `x` as safe, while in `g`, it must mark `y` as unsafe since it escapes the function and may be accessed by another thread. Compiler may store both `x` and `y` on the stack.[2] Now consider some code that calls function `f` and then calls `g`. Both `x` and `y` will be stored in the same physical location due to the runtime call stack growing and shrinking on function call and return. Furthermore, it is essential that the write to `y` complete (retire from the store buffer) after the write to `x`, otherwise we risk violating sequential program semantics even in the absence of concurrent threads.

In order to ensure that such code executes correctly, compiler is extended to emit special instructions to drain safe or unsafe store buffers selectively. At the beginning of a function, that has at least one local variable escaping the function scope (function `g` in the example), compiler inserts an instruction to drain the safe store buffer. Draining the safe store buffer ensures that all safe stores from earlier functions have updated the respective cache blocks. In the example, draining the safe store buffer ensures that write to `y` is performed after write to `x`. Similarly, at the end of function `g`, compiler also inserts another instruction to drain unsafe store buffer. Draining unsafe store buffer is performed to ensure that if a stack location (that is mapped to `y` in function `g`) is reused and is marked safe in some later functions, safe stores to that location will be performed after unsafe stores in function `g`. Draining store buffers could be expensive, but it is done only for functions in which function local variables escape the function's scope. If none of function local escape the function's scope, we do not incur any overhead. We also extended ISA to have such special instructions for draining store buffers selectively.

[2]In function `f`, the compiler might use a register for `x` and never assign it a physical memory location. Nevertheless, it is valid behavior to store `x` on the stack.

Alternatively, hardware could be extended to perform an additional check for every store. Before committing a safe store from the ROB, the processor checks the FIFO store buffer with unsafe stores for any conflicting store (a store with the same address), and vice versa. If a conflicting store is found in the other buffer, the commit is delayed until the conflicting store retires. This scenario is a rare occurrence, because it is unlikely that two function-local variables mapped to the same physical location will be of different type *and* both have stores in-flight at the same time.

This approach of extending hardware to check for conflicts on store was rejected as it requires an additional CAM lookup for each store that is committed. This expensive CAM lookup would be required for every store even though conflicts would rarely be found. In comparison, draining store buffers is effective and incurs additional overhead only in rare cases.

Additional care must also be taken with local variables of union type. For instance, notice that the address of `p` is never taken in Figure 4.3c. But, because it is essentially an alias for `i` which does have its address taken, the static analysis must classify both variables as unsafe.

4.5.3 CISC Architecture

We have so far assumed that a memory instruction in a program's binary can access only one variable. However, in the CISC architecture an instruction may access multiple variables. For such instructions, we propose to extend the ISA to provide one extra bit per memory operand in the instruction's machine code. This will allow compiler to mark each memory access in a memory instruction as safe or unsafe. A processor can use this information to classify a micro-operation generated for each memory access in a CISC instruction as safe or unsafe.

4.6 Dynamic Classification of Memory Accesses

A static technique does not have the benefit of observing the actual runtime stream of memory accesses. It must conservatively classify accesses at compile time. Therefore, a complementary dynamic technique for determining if a memory access is safe or not is discussed below. As described in Section 4.3, an access to private or shared, read-only locations is safe. To determine safe accesses, we leverage the hardware memory management unit (MMU) and the OS page protection mechanism [48, 30, 35].

4.6.1 Background: Process-Level Page Protection

Current systems provide page protection at the process level. Each process has a page table that is shared among all the threads of the process. Each page table entry contains the read and write access permissions for a page. In the execute stage of a memory operation, after its effective address is resolved, this virtual address is translated by the processor to the corresponding physical address. To assist in fast translation, the processor uses a Translation Lookaside Buffer (TLB) in each core. Each TLB entry caches a page table entry for a thread executing on its processor core. It includes read and write permission bits, which are checked by the processor when it executes loads and stores respectively. A page-fault exception is raised to the OS on detecting a permission violation. On a TLB miss for an address, a TLB miss handler (hardware assisted page-table walker) is executed, which fetches the page entry from the main memory, and allocates and initializes a TLB entry for it.

4.6.2 Proposed Extension: Thread-Level Page Protection

A page table is shared by all the threads in the process. In order to detect page sharing among threads and determine safe accesses at runtime, page table entries are extended to keep track of the sharing state for pages. Figure 4.4 shows the sharing states that a page can be in. Following fields are added to keep track of these states: (a) a thread identifier (`tID`), (b) a `Read-Only` bit, and (c) a `Shared` bit. Any access to a page in $\langle shared, rw \rangle$ state is considered unsafe, and all the others are considered to be safe.

TLB entry is also extended with an additional `Safe` bit. A processor consults this bit during address translation to determine if an access to a page is safe, and if it is, it sets the `ds` bit for the access in the ROB.

The rest of the section describes how the above states are maintained and how memory ordering constraints are guaranteed when a page changes its state.

4.6.3 State Transitions and Guaranteeing Memory Ordering Constraints

When a page is allocated by a page fault handler, its state is set to $\langle untouched \rangle$ (Figure 4.4). Its `tID` is set to `INV` to indicate that no thread has executed a read or write to this page yet. Also, its `Read-Only` bit is set and `Shared` bit is reset.

The first thread to issue a read to a page will trigger a TLB miss. The TLB miss handler checks if the page has already been allocated. If so, it checks the `tID` of the page and determines that this read is the first access. It then sets the page state to $\langle private, ro \rangle$ by setting its `tID` field. It allocates a TLB entry, sets the safe bit, but resets the write permission in the TLB entry, irrespective of the write permission bit's value in the page table entry. This allows the system to detect when the same thread attempts a write to this page, as that would cause a page fault. The page fault handler can then check the write permission for the page in its corresponding page table entry. If the attempted write is legal, the page fault handler changes the state of the page to $\langle private, rw \rangle$. Also, the write permission for the page is enabled in the TLB entry to allow future writes from the same thread. The safe bit in the TLB entry would remain set.

When another thread issues a read to a page in the $\langle private, ro \rangle$ state, it would also incur a TLB miss. The TLB miss handler determines that the page has been read by another thread, and changes the state of the page to $\langle shared, ro \rangle$. An entry in the local TLB is allocated with the safe bit set, but the write permission is disabled.

The state of a page can transition to the unsafe $\langle shared, rw \rangle$ state from three different safe states as shown in Figure 4.4. Care must be taken during these state transitions to guarantee memory ordering constraints. Let us assume a thread `P` owns a page in $\langle private, ro \rangle$ state, and a remote thread `R` issues a store to that page. The TLB miss handler in the remote processor core running `R` determines that the page needs to be transitioned into the $\langle shared, rw \rangle$ state, because the `tID` in the page table entry would be different from `R`. Before modifying the state of the page table entry, the handler issues an inter-processor-interrupt (IPI) to the processor core running

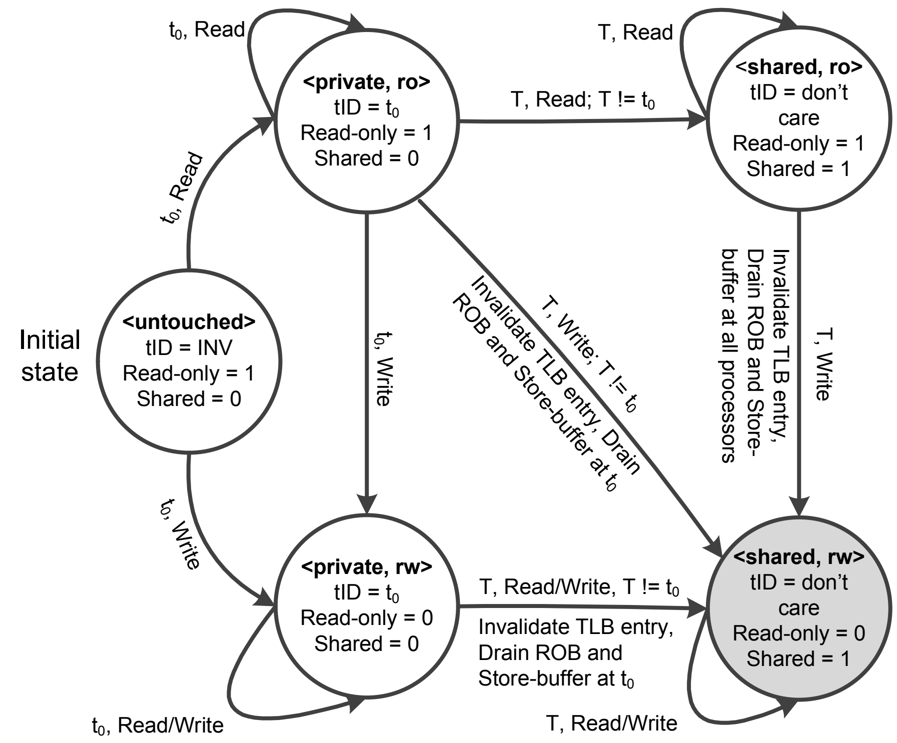


Figure 4.4: State transition of a page. Accesses to the shaded state are unsafe.

P.[3] When the processor core running P receives the IPI, the interrupt handler invalidates the corresponding entry in its local TLB, and sends an acknowledgment back to the core running R. Note that before any interrupt handler begins its execution, the processor flushes both the ROB and the store buffers in order to support a precise context switch. This behavior ensures correct memory ordering when a page transitions to the unsafe state. On receiving the acknowledgment, the TLB miss handler of R updates the state of the page to $\langle shared, rw \rangle$, and allocates a TLB entry for the page by initializing it to the permission bits in the page table entry, but resets the Safe bit. Thereafter, all accesses to the page will be treated as unsafe and ordered correctly.

[3]The TLB miss handler can determine the processor core running P by checking P's Thread-Control-Block (TCB) maintained by the OS.

The state transition from $\langle private, rw \rangle$ to $\langle shared, rw \rangle$ is handled similarly. The state transition from $\langle shared, ro \rangle$ to $\langle shared, rw \rangle$ is also similar, except that an IPI needs to be broadcast to all processor cores. To prevent races between page state updates, the TLB miss handler and the page fault handler always acquire a lock for a page before updating its page table entry.

TLB invalidations through inter-processor-interrupts could be expensive. Fortunately, this cost is incurred only once per page during an execution of a program. This allows us to provide a low-complexity hardware solution. Notice that other than the maintenance and use of the `Safe` bit, the changes required are restricted to the system software and TLB miss handler implementation.

4.6.4 Initialization Phase

Usually in a parallel program, the main thread initializes several data-structures before spawning threads. We do not want to classify a page as $\langle shared, rw \rangle$ just because it was modified by the main thread during initialization. Therefore, state of all pages is reset to $\langle untouched \rangle$ just before the main thread creates the second thread for the process.

4.6.5 Context Switches

TLB entries do not store the `tID` information. Therefore, when a thread is context switched out, the processor core cannot determine that the `Safe` bits in the TLB entries belong to the older thread. This problem of virtualizing the TLB across context switches is also a problem for supporting process-level page protection. Many processor implementations employ a simple solution that flushes the TLB entries on a context switch, which is sufficient to ensure correctness for proposed design as well. However, some newer implementations maintain additional tags in each TLB entry to efficiently support virtualization [29, 86, 10]. A similar hardware design could also allow us to support TLB virtualization while providing thread-level page protection.

4.6.6 Direct Memory Accesses (DMA)

Modern systems support Direct Memory Access (DMA) to efficiently transfer data from a slower physical device directly to main memory without involving the processor core's computational resources. However, this raises the question of what semantics the system should provide in case of a data race between the DMA transfer and concurrent accesses within the processor cores [56]. Proposed design leverages the observation made by Dunlap et al. [35] that the DMA transfer occurs between well-defined boundaries, and none of the processor cores should access the affected locations during that interval. This property can be explicitly enforced by the OS by acquiring access privileges to pages on behalf of the device and releasing them once the transaction is completed [35]. Another alternative is to temporarily change the state of pages that DMA can access to the unsafe state, and then restore the original state after the DMA transfer completes. Both of these alternatives would ensure SC even in the presence of DMA accesses. Another simpler option would be to assume that the system is properly synchronized with respect to DMA, and make no guarantee when races exist between DMA accesses and regular processor core accesses.

4.7 Evaluation

This section presents the evaluation of SC-preserving hardware. The evaluation tries to answer following questions:

- What is the performance overhead of SC hardware design when compared to TSO? What are the advantages over baseline SC?
- What is the accuracy of static and dynamic classification schemes when compared to a byte-level dynamic classification scheme?
- What is the performance overhead of guaranteeing end-to-end SC when compared to executing stock compiler's binary on TSO hardware?

4.8 Methodology

Proposed SC design is modeled using a cycle-accurate, execution-driven, Simics-based, full-system simulator called FeS2 [37]. The system models a 64-bit 16-core processor with an on-chip network. Details of the processor configuration are listed in Table 4.1. For baseline SC and TSO processor, a 64-entry FIFO store buffer with 8-byte (one word) entries is assumed. For the proposed SC design, in addition to the 64-entry FIFO store buffer, another 8-entry unordered store buffer with 64-byte (one L1 cache block) entries is modeled. The unordered store buffer allows out-of-order retirement of stores and coalesces multiple stores to the same cache block. Section 4.8.3 evaluates the sensitivity of proposed design to various store buffer sizes.

Table 4.1: Processor Configuration

Processor	16 cores operating at 4 GHz
Fetch/Exec/Commit	4 instructions (maximum 2 loads or 1 store) per cycle in each core
FIFO Store Buffer	64 8-byte entries
Unordered Store Buffer	8 64-byte entries; coalescing
L1 Cache	64 KB per-core private, 4-way set associative, 64 byte block size, 2-cycle hit latency, write-back
L2 Cache	512 KB private, 4-way set associative, 64 byte block size, 10-cycle hit latency.
Coherence	MOESI directory protocol
Interconnection	Torus-2D topology, 512-bit link width, 8-cycle link latency.
Memory	160 cycles (40 ns) DRAM lookup latency.

For all of the SC and TSO designs, in-window speculative load execution is implemented as described in [41]. For stores, exclusive prefetch [41] are also modeled which can reduce the latency of a store by obtaining the necessary write permission before the store is able to retire from the store buffer and write the cache block. TSO and SC simulations are functionally equivalent, because our front-end Simics functional simulator is SC. However, back-end timing simulator enforces the appropriate set of memory ordering constraints depending on the simulated memory model.

To implement the static classification scheme, the SC-preserving compiler is extended to classify private accesses and communicate this information to the hardware through an ISA extension. Currently, static classification is performed only for application code, because we were not able to recompile the Linux kernel and `glibc` using our compiler. Therefore, evaluation underestimates the potential benefits of the static and hybrid classification schemes.

Three variants of the proposed SC design are evaluated which are based on memory access classification scheme: static only (`SC-staticOnly`), dynamic only

(`SC-dynamicOnly`), and hybrid (`SC-hybrid`). These schemes are conservative in classifying a memory access as safe. Therefore, they may misclassify a safe memory access as unsafe. To understand the accuracy of our classification schemes, a hypothetical system was evaluated that dynamically tracked the type of a memory location at the byte granularity (`SC-ideal`), which solves the false sharing problem in page-level dynamic scheme. This fourth variant would be too expensive to realize in an actual hardware, but it is useful as a limit study.

Studied benchmarks include the Apache web server and applications from the PARSEC [15] and SPLASH-2 [117] benchmark suites. PARSEC benchmarks were run with the "simlarge" input set. For `barnes`, 65536 `nbody` was simulated. For Apache, the SURGE [14] benchmark was used to benchmark the server. For the SPLASH-2 benchmarks, complete parallel section was simulated. For Apache, caches and micro-architectural structures were warmed up for 20000 transactions, and then performance was measured for the next 20000 transactions. It was not feasible to simulate the entire parallel section for the PARSEC benchmarks due to their long execution times. Therefore, for these programs, a sampling approach was used. In this approach, five sampled checkpoints were created that span across the entire parallel section of the program . For each checkpoint, after the warmup phase (100K stores per core), timing was done for at least 10 million stores for each processor core. This sampling approach was employed for comparing hardware designs running the same binary. However, measuring progress in terms of stores may not be accurate while comparing the performance of binaries produced by two different compilers (SC-preserving and stock compiler). Therefore, for such comparisons, entire execution of

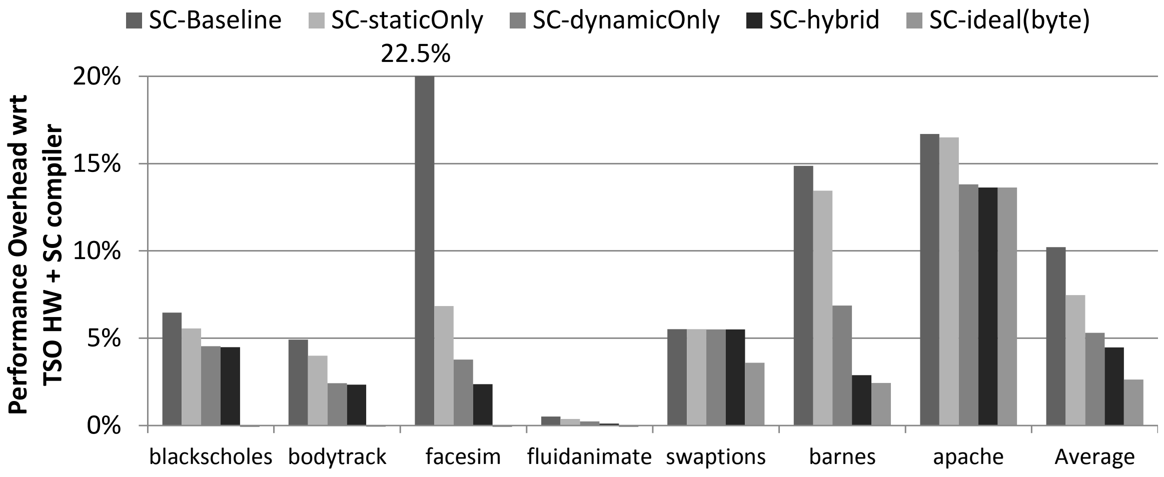


Figure 4.5: Performance of the baseline SC and variants of the proposed SC designs compared to TSO.

the parallel section was simulated. While simulating the dynamic and hybrid schemes, tracking the state of a page was started only after the parallel section starts executing. Evaluation measures the performance of both user-level and system execution in full-system simulation using instructions-per-cycle (IPC) as the performance metric.

4.8.1 Performance of Memory Access Type Driven SC Hardware

Figure 4.5 compares the performance of the proposed SC hardware to a baseline SC hardware design. The performance overhead of all configurations is shown relative to a TSO hardware design that is similar to modern x86 processor implementations. All the configurations use the SC-preserving compiler implementation. Therefore, SC hardware provides end-to-end SC and TSO hardware provides end-to-end TSO.

While proposed optimizations may not have an effect on programs that already provide good SC performance, they significantly reduce the overhead for those programs that do suffer a high performance penalty due to SC constraints. On average, SC-baseline has a performance overhead of about 10.2%. The maximum overhead

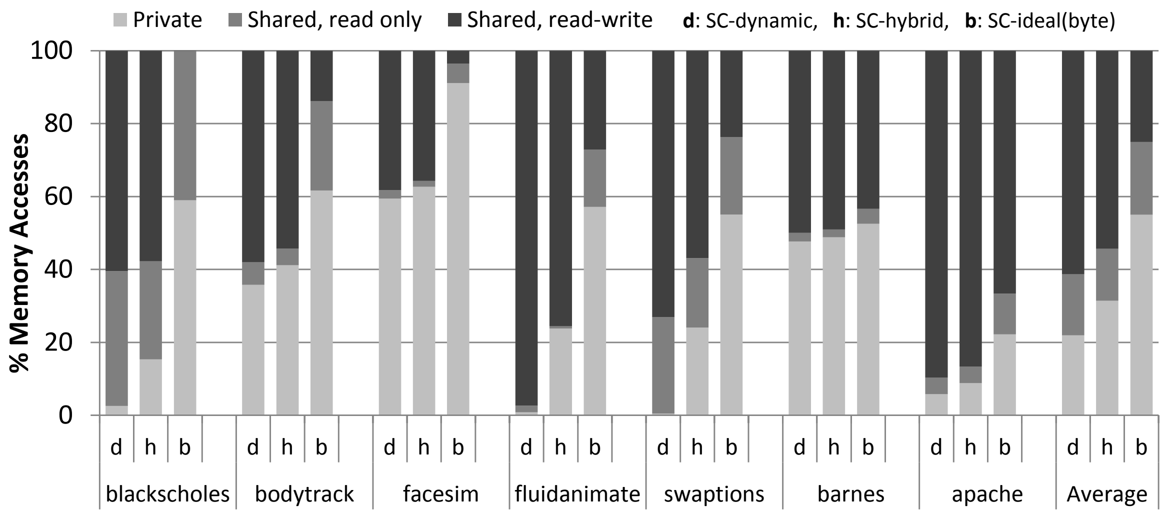


Figure 4.6: Classification of memory accesses by various methods.

for SC-baseline, however, is much higher: 22.5% (`facesim`). `SC-staticOnly` reduces the overhead to 7.5% on average, with a maximum of 16.5%. `SC-dynamicOnly` incurs only 5.3% overhead on average. The proposed SC design, `SC-hybrid`, which uses both static and dynamic classification schemes, has an average overhead of 4.5%. Worst case overhead for `SC-hybrid` is 13.6% (`apache`) which is a significant reduction from the 22.5% (`facesim`) observed for SC-baseline. The proposed design's performance is close to that of `SC-ideal` , which uses a byte-level classification scheme. We conclude that our optimizations are effective in reducing the SC memory ordering overhead when it is present.

Figure 4.6 compares the accuracy of classification schemes, which determines the effectiveness of our SC hardware optimizations. On average, our page-based dynamic scheme (`SC-dynamicOnly`) classifies 38.7% of memory accesses as safe. Combining this with the static scheme improves the accuracy further to 45.6%. Classifying access at byte granularity (practically not feasible to implement) reveals that 75.0% of total memory access were to safe locations.

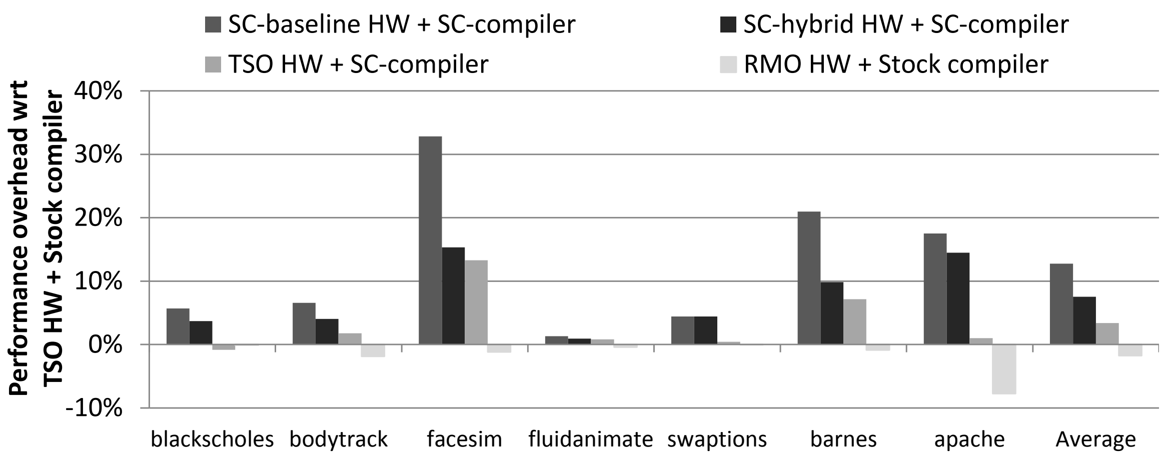


Figure 4.7: Comparison of our proposed system with baseline SC, TSO, and RMO hardware.

Applications with a higher proportion of safe accesses (96% for `facesim`) benefit significantly from proposed optimization as discussed earlier. However, proportion of unsafe accesses is not the only factor that determine the final SC hardware overhead. Cache miss rate of unsafe accesses have a more direct influence. For example, `SC-hybrid`'s overhead for `bodytrack` is lower than that of `swaptions` even though they have about similar fraction of unsafe accesses. This is because `bodytrack`'s has lower cache miss rate than `swaptions`.

4.8.2 Cost of End-to-End SC

The cost of end-to-end SC is shown in Figure 4.7. Baseline is chosen as a TSO processor running the binary produced by the stock LLVM compiler as it represents the most commonly used systems today. End-to-end SC has two sources of overhead: 1) the cost of preserving SC in the compiler, and 2) the cost of enforcing SC in the hardware. In Figure 4.7, we observe that the cost of preserving SC in the compiler is on average about 3.4% (`TSO HW + SC compiler`). If baseline SC hardware is used, the total end-to-end SC cost is about 12.8% on average. However, by using the hybrid classification scheme, proposed SC design reduces the cost to 7.5% on average. This overhead is only slightly higher (9.2%) when compared to a relaxed memory model (RMO) hardware (which is sufficient to support C++ or Java memory model) executing the stock LLVM compiler's output.

4.8.3 Sensitivity to Store Buffer Sizes

`SC-hybrid` design assumed an additional unordered store buffer when compared to the `SC-baseline`. When the size of the two store buffers was halved in `SC-hybrid`, which made them area neutral with the store buffer in `SC-baseline`, the increase in performance overhead was negligible (less than 1%).

It is important to note that for a store buffer, the dominant cost is not area, but rather the latency and power cost of associative lookups. An associative lookup of the store buffer is necessary for each load to support store-to-load forwarding. In our design, a load has to search only one of the two store buffers. Thus, the additional unordered store buffer in SC design does not aggravate this dominant overhead.

4.8.4 Processors with Limited Instruction Level Parallelism

In earlier sections, overhead of end-to-end SC was discussed for out-of-order issue processors. It also interesting to measure this cost for processors with limited ILP (e.g in-order cores). Because of limited ILP, in-order issue cores have very limited opportunity to hide the cache miss latency. Figure 4.8 presents the overhead of end-to-end SC for in-order processors. With baseline SC implementation, cost of end-to-end

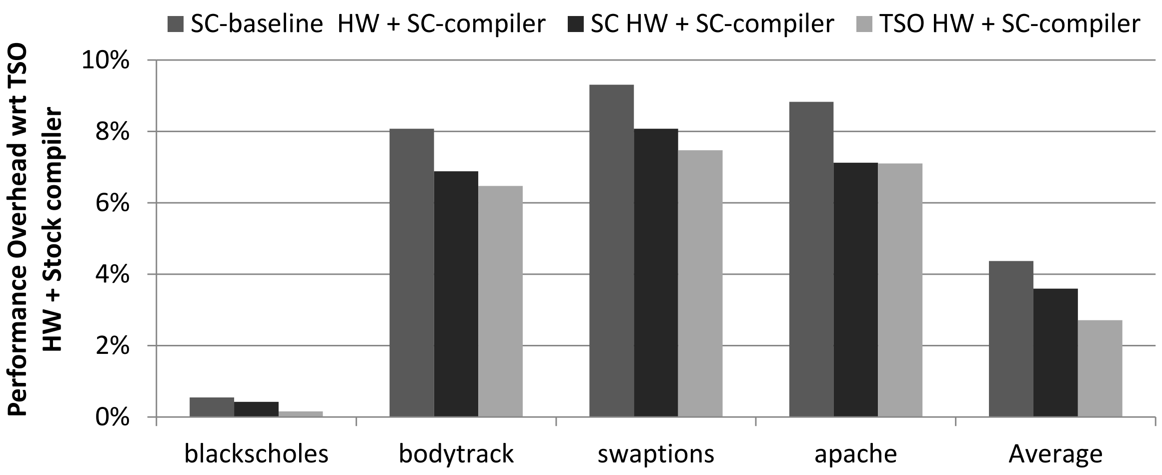


Figure 4.8: Comparison of our proposed system with baseline SC, TSO, and RMO hardware with ***in-order cores*** .

providing SC is about 6.1% compared to a system with `TSO HW + Stock-compiler`. Proposed `SC-hybrid` design reduces this cost to 4.9% on average. In the case of in-order cores, a core is not able to exploit the relaxation offered by relaxed memory models because it gets stalled owing to low available ILP. Different memory models, therefore, are likely to not exhibit significant performance difference.

4.9 Conclusion

The research in this chapter exploited an important opportunity that has been overlooked in the past while designing SC hardware: no memory model constraints need to be enforced on accesses to private locations and shared, read-only locations. By exploiting this observation, a low-complexity SC hardware design is derived that obviates the need for aggressive speculation to obtain high performance. It uses a combination of static analysis and the page protection mechanism to identify safe accesses and relax SC constraints on them. Apart from an additional unordered store buffer, there is very little hardware modification needed to support our design. Our end result is promising: SC hardware is only 4.5% slower than TSO, and end-to-end SC costs only about 7.5% when compared to the performance of a state-of-the-art compiler and TSO hardware.

For the SC memory model to be adopted at the language level, all the compilers and processors that support the language should be made SC-preserving. While our study considered one of the most widely used processor designs as baseline (an out-of-order TSO processor), further study is needed to understand the overhead due to end-to-end SC in other classes of systems (e.g., a low power in-order architecture may be important for embedded systems).

CHAPTER V

Efficiently Enforcing Strong Memory Ordering in GPUs

Earlier chapters have targeted a shared memory parallel systems built out of multi-core CPUs. This chapter focuses on Graphics Processing Units (GPUs) that have emerged as a new platform for parallel computing and are significantly different from traditional multi-cores. GPUs are now deployed across a wide range of systems that vary from mobile platforms to super computers. Modern GPUs are no longer limited to graphics applications, and are also being used as data parallel accelerators for general purpose programs. Programming models such as CUDA [87] and OpenCL [81] have enabled developers to exploit data-parallelism in general-purpose applications using GPUs (referred as GPGPU applications). Most of these GPGPU applications are not purely data-parallel and involve inter-thread communications. Ensuring correctness of these applications in the presence of wild shared-memory communication remains a challenge. This chapter explores how strong memory ordering constraints can be efficiently supported in GPUs.

5.1 Introduction

As is the case with any shared-memory multi-processor system, complexity of GPU parallel programming critically depends on the memory model of the GPU platform. Conventionally it is believed that weak memory models are suited for graphics applications. However, for GPGPU applications, this assumption may not be entirely true. Unfortunately, GPU vendors have largely ignored the need for formally defining the memory model semantics of their devices. Even today, GPU vendors' programming guides only provide informal descriptions about their memory model, leaving programmers with no choice but to rely on folklore assumptions. Not surprisingly, researchers are starting to find some serious mismatches between the memory model assumptions commonly made by the GPU programmers and what the hardware implementations end up providing [8].

Recently, there have been efforts to standardize OpenCL's [82] memory model based on C11's [62] data-race-free-0 (DRF-0) [4] model. It guarantees sequential consistency (SC) for data-race-free programs, but does not provide any semantics for programs with data-races. Researchers have refined this model for CPU-GPU devices by defining heterogeneous data-races [57], which is based on a revised happens-before model that accounts for different thread-visibility scopes for memory operations.

Before adopting a weaker memory model (DRF-0) for GPUs, it is imperative that we understand the costs of a stronger memory model such as TSO and SC for GPUs. A recent study investigated the cost of various memory models in a high-throughput processor with several in-order cores [52]. However, for a GPU architecture, several questions remain unanswered, which we seek to answer in this paper.

- What are the architectural mechanisms necessary in GPUs to enforce various types of memory ordering constraints (DRF-0, TSO, SC)? What are their performance costs?

- What is the interplay between the common GPU architectural design choices such as warp-scheduling and write-through caches, and the memory ordering constraints?

- Prefetching and in-window speculation [41] work remarkably well for CPUs. Are they feasible in GPUs? Are they equally effective in mitigating the performance overhead of memory ordering constraints in GPUs?

- How do we engineer a GPU-specific optimization for reducing strong SC memory ordering constraints?

A GPU core is typically assigned to tens of warps, where each warp contains as many threads as the number of SIMD lanes. Memory ordering constraint (RMO, TSO, or SC) can be naïvely implemented in GPUs by restricting the warp scheduling policy. To support SC, a core can issue a memory operation from a warp only if all of its preceding memory operations from the warp have completed. If a warp is stalled due to a memory ordering constraint, the core can issue instructions from another ready warp. Thus, warp-level-parallelism is effective in reducing memory ordering stalls. Furthermore, GPU's in-order core is limited in its capability in exposing intra-thread memory-level-parallelism. In GPUs with write-back caches, these two factors combine together to effectively nullify the performance overhead of SC as compared to RMO (which supports DRF-0 model) for a majority of applications we studied. Surprisingly, for some applications, SC outperformed RMO. We analyze this anomaly in depth in Section 5.6.1.

However, for a few challenging applications, overhead of SC is quite significant (maximum is ˜3x). We also found that overhead of TSO was also significant in these applications (maximum is 88%). When we employed store-buffer optimization, maximum TSO overhead reduced to about 50%. This optimization relied on one large store-buffer per warp – a significant area cost (48 KB). Given that the TSO overhead is still significant for the challenging applications, we do not see a compelling case to weaken the SC guarantees and choose TSO.

Replacing write-back caches with write-through caches eliminated the performance gap between SC and RMO for all applications, including the challenging applications we discussed above. We analyze the reasons in Section 5.6.5.

For GPUs with write-back caches, we need an efficient solution for reducing the SC overhead for the challenging applications noted above. In-window speculation [41] is inapplicable for simple GPU cores. Prefetching [41] cache blocks while waiting to resolve memory ordering constraints is a cheaper alternative, and is commonly employed in CPUs. However, to our surprise, we found that it is ineffective in reducing SC or TSO overhead, and in some cases it even reduced the performance. The reason is that, in a GPU core that multiplexes between numerous threads, there are too many premature prefetches, where a prefetched cache block gets evicted before it is used, that increase contention in memory hierarchy and can also evict useful blocks.

To reduce the SC overhead for applications with high overhead, we devise a non-speculative design based on Shasha and Snir's observation that memory ordering constraints can be relaxed for accesses to locations that are private to a thread or are shared in read-only fashion [103] (referred as safe accesses and locations respectively). We follow the memory access classification scheme used in Chapter 4.6, but at a much larger granularity (16KB). We can afford a larger granularity because data accessed by a GPU core typically exhibit very high spatial locality [93].

Despite similarities in memory access classification schemes, significant micro-architectural differences between GPUs and CPUs necessitate building a GPU specific solution (Section 5.4). These differences pertain to virtual memory support, instruction execution, number of concurrent threads per core (SM for GPU), presence of a shared memory in SM and partitioned address space, etc. We show that our proposed solution eliminates almost all the SC overhead in the applications we studied.

5.2 Background

5.2.1 GPU Architecture

Computation on GPU consists of functions or kernels that have a hierarchy of threads (Figure 5.1). Following the CUDA terminology, this hierarchy comprises a grid of thread blocks (or NDRange of work-groups in OpenCL terminology). Each thread block contains one or more warps (sub-groups in OpenCL), which is a hardware specific grouping of scalar threads (work-items in OpenCL) that execute in a lock step manner.

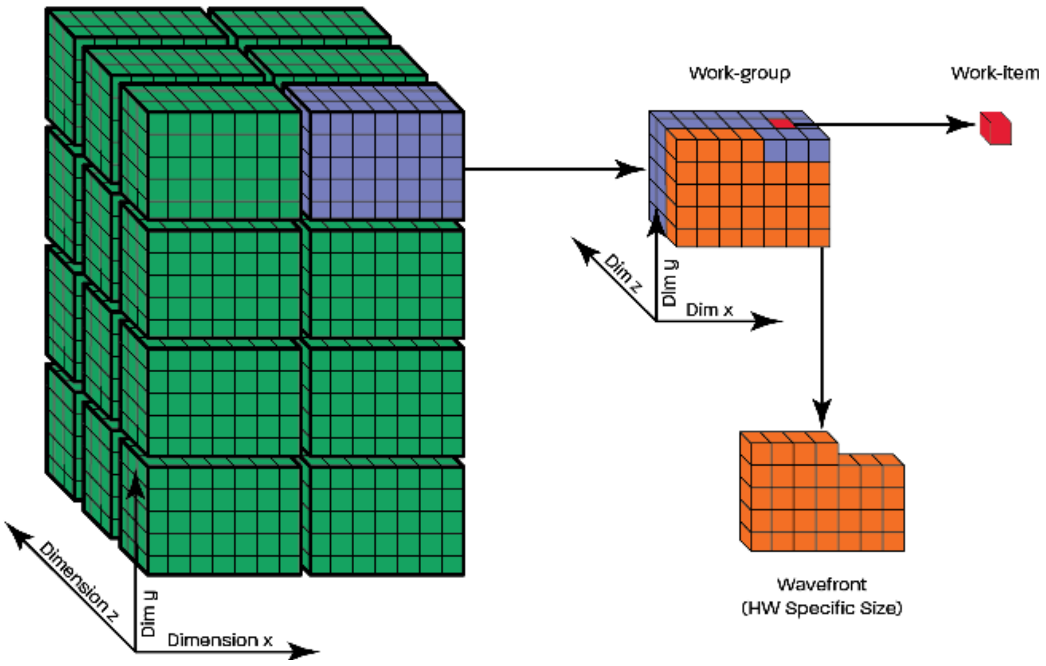


Figure 5.1: Organization of threads (work-item) in thread block (work-group) and grid (NDRange) in GPU kernel. Image is borrowed from HSA Whitepaper [67].

Memory space in CUDA follows this thread hierarchy and is partitioned with different levels of sharing among the threads. CUDA provisions for per thread *local* memory space which is private to a thread. Threads within a block cooperate by sharing data using *shared* memory space. Finally, CUDA also provisions for *global* and read-only *constant* and *texture* memory spaces which are visible to all threads in a kernel. While texture memory space is specific to CUDA, local, shared, global and constant spaces are termed private, local, global and constant in OpenCL respectively.

A GPU includes a small number of streaming multi-processors (SM) (Figure 5.5). Each SM has a large number of in-order compute cores, multiple load/store units and special function units. An SM executes instructions from a thread in the program order. A memory operation is issued to L1 cache when it reaches at the head of the memory pipeline and is then removed from the pipeline. Thus, a cache miss does not block the memory pipeline and subsequent memory operations can be issued to L1 cache as they reach at the head of the memory pipeline. A MSHR explicitly keeps track of all outstanding cache misses in an SM.

Each SM houses a shared memory, a private L1 data cache and a constant cache. All SMs on a GPU share a L2 data cache. L1 cache holds data from local and global address space. Current GPUs (NVIDIA Fermi [88], AMD GCN [11]) do not support coherent L1 data caches. Lack of hardware support for coherence burdens the programmer to explicitly manage coherency of global data that is shared among threads. Programmers are expected to add a *volatile* keyword (in CUDA) that disallows caching of such global data in L1 cache. Inability to cache global data hurts all memory models when an application exhibits temporal locality for this data (i.e. memory accesses are non-streaming type).

In contrast, ARM's Mali GPU [106] supports coherent L1 caches. Recent hardware proposals [105, 51] have also advocated for coherent L1 caches in GPUs. Future GPGPU applications could be expected to have higher communication among threads than what is present in today's data parallel applications. Simply disallowing caching of shared global data may not be an optimal approach for these GPGPU applications that exhibit temporal and spatial locality in their access pattern. All memory models benefit from hardware coherence as it enables private L1 caches to cache global data that is shared among threads. The choice of a memory model may impact how hardware cache coherence support is implemented; with relax memory models affording more flexibility in the implementation. In our subsequent discussion we assume that our baseline GPU implements a cache coherent memory hierarchy.

Table 5.1: Synchronization functions in GPGPU programming models. Notations: S: synchronization, R: read, W: write

(a) OpenCL

Synchronization operations (S)	Memory ordering parameter	Ordering enforced	Visibility scope
Fence, Atomic Operation	relaxed	—	Work-group, Device, System
	acquire	`S` → `R`; `S` → `W`	
	release	`W` → `S`; `R` → `S`	
	acq_rel	`S` → `R`; `S` → `W` `R` → `S`; `W` → `S`	
	seq_cst	`S` → `R`; `S` → `W` `R` → `S`; `W` → `S` (& total order on accesses)	

(b) CUDA

Fence	—	`R` → `R` (code: $R; S; R$) `W` → `W` (code: $W; S; W$)	Thread block, Device, System
Atomic function	—	—	Device

5.2.2 CUDA and OpenCL Memory Consistency Models

GPGPU programming models such as CUDA and OpenCL support weak consistency models based on data-race-free-0 (DRF-0) [4] memory model. These memory models match weakly ordered [34] memory models supported by GPU hardware. Under DRF-0, a SC execution is guaranteed only if program is data-race free. Current GPGPU programming models support two kind of synchronization operations: fences and atomic operations. These synchronization operations can be used to construct high level synchronization primitives.

Table 5.1 lists the synchronization operations provided under OpenCL [82] and CUDA programming models [87, Section B.5, B.6 and B.12]. A synchronization operation in OpenCL has an argument "Memory ordering parameter" which decides the memory ordering constraints enforced by the synchronization operation. The "Ordering enforced" column lists memory ordering constraints enforced by a synchronization operation for a given memory ordering parameter. There are four possible orderings for loads and stores with respect to a synchronization operation (S): $\texttt{S} \rightarrow \texttt{R}$, $\texttt{S} \rightarrow \texttt{W}$, $\texttt{R} \rightarrow \texttt{S}$ and $\texttt{W} \rightarrow \texttt{S}$. The ordering $\texttt{S} \rightarrow \texttt{W}$ indicates that later stores (in the program order) cannot complete before an earlier synchronization operation. In OpenCL, default memory ordering constraint for a synchronization operation is *memory_ order_ seq_ cst* which guarantees that memory accesses before and after the synchronization operation are not reordered with it and there exists a total order on all accesses with this constraint. To support weaker ordering constraints for low level atomic operations, OpenCL allows programmers to specify relaxed constraints such as acquire, release and acq_rel. Note that acq_rel provides read and write ordering relative to the variable, whereas, seq_cst provides read and write ordering globally.

CUDA programming model also supports fences and atomic functions to facilitate synchronization among threads, but the memory ordering constraints associated with them are very different from OpenCL. A fence in CUDA only enforces $\texttt{R} \rightarrow \texttt{R}$ and $\texttt{W} \rightarrow \texttt{W}$ memory ordering when these two reads or writes are separated by the fence. This constraint is different from either acquire or release constraints. Furthermore, atomic functions (read-modify-write operations) in CUDA do not have any memory ordering constraints associated with them.

The last column in Table 5.1 shows different visibility scopes that can be associated with synchronization operations. The visibility scope of a synchronization operation governs which threads observe its effects. Effects of a synchronization operation with device visibility scope are visible to all threads running on a GPU device. On the other hand, the effects of a system scope synchronization operation are visible to threads in the current GPU device, host CPU and other compute devices present in the system. Finally, thread block/work-group scope limits the effects of a synchronization operation to a thread block/work-group of threads only.

Achieving SC in OpenCL and CUDA: In order to guarantee SC, OpenCL requires that all data-races be marked as synchronization operations and recommends that programmer use *memory_ order_ seq_ cst* and system visibility scope for synchronization operations. While achieving SC with relaxed ordering constraints or smaller visibility scope is possible, such optimizations require significant care on part of the programmer and are highly prone to errors. On the other hand, the fence primitive alone as supported in CUDA is not sufficient to build high-level synchronization operation because it does not enforce $\mathtt{S} \rightarrow \mathtt{R}$ (for acquire) or $\mathtt{R} \rightarrow \mathtt{S}$ (for release) ordering constraint which is required for protecting critical sections. To enforce these missing memory ordering constraint, an atomic operation (a read-modify-write operation) could be used before or after the fence instruction for acquire and release semantics respectively.

5.3 Enforcing Memory Ordering Constraints in a GPU

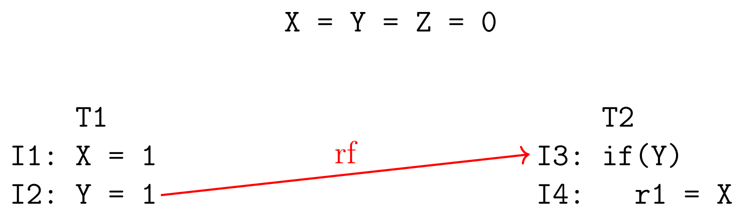
In this section we discuss mechanisms for enforcing memory model constraints in GPUs. First, we discuss when and how two memory accesses can appear to have executed in an order different from their program order. Then, we describe how different memory models can be supported in hardware by controlling how warps are executed in an SM. Finally, we also describe how a memory model may affect other architectural features like warp scheduling.

5.3.1 Violation of Ordering Constraints

In-order execution does not imply in-order completion of memory operations. Recall that a memory operation that misses in L1 cache does not stall the execution. Therefore, a thread could have multiple memory requests outstanding. Two memory requests in out-of-order fashion for variety of reasons: data caching at L1 cache, interconnect delivering requests in out-of-order manner, requests going to different partitions. However, in context of memory models, we are interested in how reordering of access can be observed by other threads.

Reordering of memory accesses in one thread could be visible to another thread running on different SM for following reasons:

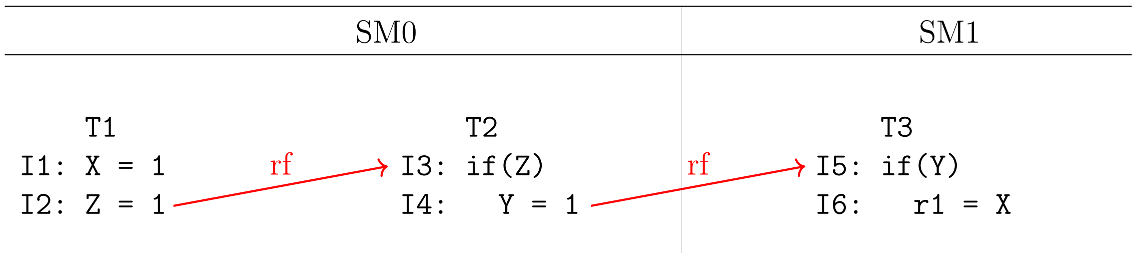
Caching at L1 data cache: In a cache coherent system, a thread can observe reordering of memory accesses in a thread running on different SM. Figure 5.2a shows an example wherein two threads $T1$ and $T2$ are synchronizing with each other. Assume that $T1$ and $T2$ are running on streaming processors $SM0$ and $SM1$ which are



```
                X = Y = Z = 0

      T1                              T2
I1: X = 1                        I3: if(Y)
I2: Y = 1                        I4:   r1 = X
```

(a) Direct synchronization



```
      T1                  T2                  T3
I1: X = 1            I3: if(Z)           I5: if(Y)
I2: Z = 1            I4:   Y = 1         I6:   r1 = X
```

(b) Indirect synchronization

Figure 5.2: Direct and indirect synchronization. rf is *reads-from* relation that associates a read with a unique write that supplies the value. Locations X and Y are in global memory and Z is in shared memory.

caching Y and X respectively in their L1 cache. Even though $T1$ executes $I1$ and $I2$ in program order, $I2$ completes before $I1$ as $I1$ misses in L1 cache. The read in $I3$ *reads-from* $I2$ and returns 1. At this time, $T2$ is ready to execute $I4$, but there is no guarantee that $I1$ has been performed yet; $I4$ returns an old value of X. Thus, $T2$ has observed the out-of-order completion of access in $T1$.

Reordering in interconnect: The shared L2 cache in a GPU is typically partitioned into multiple banks. If GPU interconnect delivers two requests from different SMs, but to same L2 bank in an order different from their issue order, reordering of memory access could be visible across different SMs. Continuing with our earlier example, assume that X and Y map to different L2 banks and are not cached in any L1 cache. $T1$ executes two stores which are sent to their L2 banks after missing in L1 cache. In meantime, $T2$ executes $I3$ and issues a read request to L2 cache. Assume that $I3$ *reads-from* $I2$. After the branch in $I3$ has been resolved, $T2$ executes $I4$ and issues a read miss request to L2 cache. If the read request for X reaches the L2 bank containing X before the write request for X (possibly due to different latency to reach the L2 bank), $T2$ will end up with an old value of X, exposing out-of-order completion of accesses in $T1$.

Current NVIDIA and AMD GPUs [89, 11] employ a crossbar as their interconnect. A crossbar interconnect does not reorder two requests going to the same destination. However, if future GPUs adopt a mesh or other interconnects, such reordering of cache requests would make out-of-order completion of memory accesses visible to other SMs. In our designs, we do not rely on such ordering guarantees from the interconnect as we explicitly keep track of pending accesses.

Multiple address spaces: Both CUDA and OpenCL provide partitioned address spaces to their programs. Current GPUs have distinct pipelines for different address spaces; accesses from different address space are unordered. Threads synchronizing via accesses in different address spaces need to use appropriate fences to achieve correct synchronization. For example, Figure 5.2b shows an execution in which two threads $T1$ and $T3$, running on different SM synchronize indirectly via thread $T2$. Assume both $T1$ and $T2$ are running on same SM and synchronize through accesses to Z which is in *shared memory.* $T2$ and $T3$ synchronize through accesses to Y in global memory space. It is possible for $I6$ to read an old value of X in this program. Under SC, $I1$ happens-before $I6$, whereas, this example contains a heterogeneous race [57] under weaker memory models and requires either fences with device or larger scope or sequentially consistent atomic accesses for Z and Y (in OpenCL).

Memory reorderings not visible to threads on same SM: So far, we have discussed how one thread could observe reordering of memory accesses in another thread executing on a different SM. However, when threads are running on same SM, out-of-order completion of memory accesses in one thread is not visible to other threads. This is because threads running on same SM share memory hierarchy (e.g. shared memory and L1 data cache). Therefore, L1 cache and shared memory act as serialization points for all global and shared memory accesses respectively within an SM. Since each thread issues memory operations in the program order, the serialization order of accesses at L1 cache is consistent with the program order. Therefore, a cycle cannot exist in the total order of memory accesses from an SM.

To see why this is the case, consider the example shown in Figure 5.2a where two threads $T1$ and $T2$ are running on same SM. First, $T1$ issues cache request for $I1$ and $I2$ in program order. Afterwards, when $T2$ executes $I3$ it returns the value written in $I2$. Therefore, when $I4$ is sent to L1 cache, it is guaranteed to return a value that is written by either $I1$ or a later access because $I4$ is serialized after $I1$ at L1 cache. Therefore, $T2$ is not able observe out-of-order completion in $T1$. This is also true when either one or both locations are in shared memory.

This claim is true only if GPU guarantees SC per location. This principle ensures that value taken by a location are the same as if on SC. Nearly all CPU models guarantee this except SPARC RMO [113, Chap. D.4]. NVIDIA GPUs are known for not guaranteeing read-read coherence for a single location [8]. Therefore, two consecutive reads might return inconsistent values if they have different *cache operators* provided in CUDA. These cache operators specify if a memory access cache the data at different cache levels or not.

In our baseline system model, we assume that it guarantees SC for accesses to a single location. Therefore, two consecutive reads to a single location in a thread must return consistent values. This is why $I4$ must observe the value of $I1$ because $I4$ is issued after $I1$ and both of them access the same memory location.

5.3.2 Relaxed Memory Ordering

Relaxed memory models mandate memory ordering constraints for synchronization operations which then can be used to order execution of memory accesses. Furthermore, as described in Section 5.2.2, synchronization operations have concomitant visibility scope which mandates to which threads their effects are visible. There is very little public information on synchronization operations are implemented in modern GPUs that support RMO. Following text describes our way of supporting synchronization operations in our baseline GPU.

Accesses to local address space do not require any ordering constraint because they are private to each thread. Thus, we only have to worry about accesses to shared and global address space. Thread block/work-group visibility scope is naturally supported in GPU hardware because all threads from a thread block are executed in one SM and reordering of memory accesses in a thread is not *observable* in other threads running on *same SM* because all threads share same path to memory hierarchy. While conveying synchronization operations with thread block visibility seem redundant from hardware perspective, they are used by GPU programmers to prevent compiler re-ordering of memory accesses.

Other visibility scopes such as device or system require that a synchronization operation's effect is made visible to all threads running on a device. This can be done by maintaining two per warp counters (for loads and stores) to keep track of pending accesses to global memory address space. Then, based on the memory ordering constraint desired by the synchronization operation, its issue can be stalled till relevant counters become zero. Delaying issue of a synchronization operation is preferable over allowing it to execute and then stalling the memory pipeline. Such delaying prevents this warp from interfering with other warps that share the memory pipeline. Similarly, the aforementioned counters are per warp to avoid a synchronization operation in a warp from interfering with operations of other warps.

To support indirect synchronization (Figure 5.2b), accesses to Z and Y needs be marked as synchronization operations with device scope. This will ensure that before a synchronizing shared space access is executed, preceding global accesses in its thread have been performed.

5.3.3 Total Store Order

Today, most common desktop and server processors support total store order (TSO) memory model which provisions for store buffer allowing loads to bypass preceding stores. The key insight behind TSO is that loads are more critical for performance and they should not be delayed by pending stores which are off the critical path. For desktop and server workloads, TSO represents a good design point and we investigate its efficacy for GPUs.

The naïve approach to support TSO would be to leverage per warp counters to keep track of pending loads and stores. Based on values of these counters, loads can be stalled at issue for preceding pending loads, while stores can be stalled at issue for preceding pending loads and stores. Stalling at issue, while preferable, will cause loads following a delayed store to also get delayed. This unnecessarily constrains TSO.

For such situations, CPUs typically employ a store buffer that holds stores which are not yet ready to be issued to cache due to memory ordering constraints. This optimization allows processors to commit stores from reorder buffer (ROB) and allow later loads to get completed. Thus, to avoid stalling of load operations due to an earlier store, we can extend the naïve TSO design with per warp store buffers.

In our evaluation, we find that TSO does not provide any significant performance advantage (Section 5.6) over SC. Our naïve SC design, which we describe next, matches the performance of TSO for most benchmarks. For rest of the benchmarks, both SC and TSO perform poorly in comparison to RMO. While TSO has small performance advantage over SC, we believe this advantage is often not large enough to warrant weakening of concurrency semantics.

5.3.4 Sequential Consistency

While SC is the most constrained of all memory models we discuss, it provides concurrency semantics that naturally match with programmer's intuition. This boosts programmability which makes SC an attractive choice for CPUs [55].

SC requires that loads and stores appear to have completed in the program order. It could be trivially achieved by utilizing per warp counters for pending loads/stores to global memory space. To support indirect synchronization (Figure 5.2b), shared memory accesses wait for preceding global memory accesses to get completed before they can be executed. Though simple in nature, we find that this design performs very close to RMO for most of the benchmarks that we study (Section 5.6). The primary reason for effectiveness of this naïve design is availability of warp-level-parallelism (WLP) in GPGPU applications.

Unlike CPU applications, where typically only a very small number of threads are executing simultaneously in any core, GPGPU applications have a large number of warps concurrently executing in an SM. Due to large number of available warps, an SM is able to issue memory accesses from different warps to hide overhead of SC ordering constraints. Even though a single warp would observe high cost of SC ordering, an SM incurs only a small overhead.

Although higher WLP helps SC by having multiple warps to issue memory accesses from, it could also result in poorer SC performance. This is because increasing the number of threads per SM typically reduces available per thread cache capacity and this could result in higher cache miss rates. While high cache miss rates will affect performance of both SC and RMO, if the application has intra-thread MLP, RMO can exploit that by overlapping cache misses thus hiding their latency. However, even RMO is constrained in completely exploiting intra-thread MLP due to in-order execution in GPU which stalls an instruction if it has unsatisfied data dependencies.

Figure 5.3 shows an example that depicts this scenario. In this example three independent loads L_1, L_3, and L_4 can be issued in parallel by an out-of-order processor, but L_2 cannot be issued before L_1 completes due to its true dependency on L_1. Due

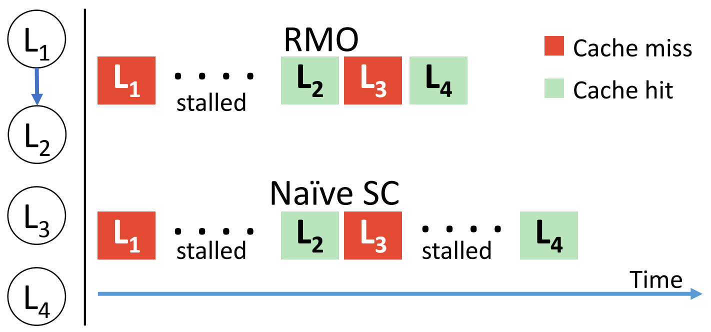


Figure 5.3: Utility of relaxed ordering constraints in an in-order GPU.

to in-order execution, loads L_3 and L_4 also get stalled until L_2 is executed. Therefore, even RMO may not able to exploit all intra-thread MLP available in a program.

However in absence of such data dependency induced stalls, RMO can issue multiple memory requests, whereas SC can issue just one request per thread. This inadequacy of SC is exhibited as huge performance overhead (up to 3x) for SC in some of our experiments.

5.3.5 Common Memory Ordering Optimizations

In-window speculation [41] is commonly employed in CPUs to reduce TSO and SC overhead, but it is not suitable for GPUs as execution window contains only small number memory instructions from a warp.

Prefetching [41] is another common optimization to reduce the cost of memory ordering overhead. While a memory access is waiting for a memory ordering constraint to be resolved, its cache block can be prefetched. We can support this optimization in GPUs by extending our naïve SC design with a per warp FIFO buffer to hold memory accesses while they wait for their ordering constraints to get resolved. Meanwhile, an SM can issue prefetch requests for these waiting memory accesses to lower the miss latency. As we will see in Section 5.6, this does not help much in bringing down the overhead of SC in GPUs because an SM switches frequently between different warps, and many prefetched blocks end up getting evicted before they are used.

5.3.6 Impact of GPU Architectural Features

We have discussed how threads issue individual memory operations under different memory models. Here we discuss how other common GPU architectural features impact the cost of memory ordering constraints.

5.3.6.1 Warp Scheduling

A warp scheduling policy's performance is greatly affected by memory model because the memory model determines when a thread can execute memory operations. This interplay of thread scheduling and memory model is largely absent from CPU memory model tradeoffs because each core has only small number of concurrent threads, often one or two.

The simplest warp scheduling policies include round-robin and greedy-then-oldest (GTO). Under GTO, a warp is run until it stalls and then the oldest ready warp is selected to execute next. Current GPUs use warp scheduling policy similar to GTO [80]. Recent proposals [98, 99, 64, 63, 102] improve upon these simple scheduling policies by leveraging intra-thread MLP and weak consistency model of GPUs. Stronger memory model implementations (e.g. naïve SC or TSO) could reduce efficacy of these proposals by restricting threads from exploiting intra-thread MLP. Nevertheless, optimizations that allow threads to expose intra-thread MLP while preserving

```
for (each edge e of a graph node)
 if (!g_graph_visited)
   g_cost[e] = g_cost[tid] + 1;
```

(a) Streaming stores in `bfs`.

```
for (k=0; k < NK; k++) // NK = 512
  c[tid] += ALPHA * a [i*NK+k] * b[k*NJ+j];
```

(b) Non-streaming stores in `gemm`

Figure 5.4: Example of streaming and non-streaming stores in GPGPU programs

SC could regain most of the benefits of these proposals. Section 5.4 describes one such implementation of SC in GPU.

5.3.6.2 Cache-Write Policy

Current GPUs typically include write-through L1 caches with no-write allocate policy. To understand how memory ordering constraints are affected by cache write policy, we evaluate performance of different memory models on a cache coherent GPU system with write-through L1 caches that use no-write allocate policy. Performance impact of cache write policy depends on store locality present in the program. If a large fraction of stores are streaming (i.e., no reuse, e.g., Figure 5.4a), no-write allocate policy results in performance improvement due to an increase in effective cache capacity for loads. However, if a majority of stores are non-streaming (e.g., Figure 5.4b), sending all writes to L2 cache will increase pressure on memory bandwidth and could result in a performance loss.

Performance gap between RMO and SC is expected to be higher for write-through caches [90], but our evaluation of GPGPU programs shows evidence contrary to this expectation (Section 5.6.5).

5.4 Efficient SC for GPU

This section describes an optimized non-speculative solution to support SC in GPUs while achieving performance close to RMO. First, it provides an overview of key ideas behind this solution, then discusses implementation details.

5.4.1 Overview

To improve performance of the naïve SC design, we again exploit the observation that memory ordering constraints can be relaxed for accesses to locations that are either private to a thread or shared with other threads in read-only fashion [103]. Memory reordering of such accesses cannot be observed or altered by other threads. We refer to accesses with this property as safe accesses and rest as unsafe in the following discussion.

Safe accesses can be freely reordered with other accesses provided intra-thread data dependencies are preserved. Unsafe accesses, however, are completed in the program order to ensure SC. To ensure correctness, accesses must be classified as either safe or unsafe in their program order.

We can identify safe accesses by observing thread-level data sharing at runtime. First, we classify memory locations at word granularity as safe or unsafe; then accesses are classified as safe if they access safe locations (Section 4.6). GPGPU applications typically exhibit high spatial locality [93] in memory accesses. We exploit this observation by keeping track of sharing of memory locations at a large granularity (referred to as sectors), thereby saving required hardware space.

Recording sharing information at thread granularity is prohibitively expensive in GPUs because of huge number of concurrent threads in GPUs. To keep hardware cost under control, we record this information for each SM instead of each thread. This gives us a set of potentially safe accesses (referred to as SM-safe). Similarly, shared memory accesses can also be treated as SM-safe accesses. These accesses are only potentially safe because multiple threads from same SM could be performing conflicting accesses to an SM-safe location. We treat SM-safe accesses as "safe" accesses and relax ordering constraints for them, but enforce strong ordering on conflicting accesses to an SM-safe location.

Classification of memory accesses: In CPUs, accesses are classified during virtual to physical address translation [48, 30, 104] by leveraging page tables and translation lookaside buffer (TLB). Recent GPU virtual memory proposals [93, 91] also provide a CPU like virtual memory support for GPUs. These proposals can be extended to also provide classification of accesses during address translation stage.

Although current GPUs implement virtual memory support, there is very little public information about them [31, 109]. It is not clear whether address translation is carried out prior to L1 cache access or not. Intel and AMD GPUs have Input Output Memory Management Units (IOMMU) [58, 9, 12] equipped with their own page tables and TLBs, but these IOMMUs are placed in memory controller and are accessed only after a miss in L1 cache. In this paper we have assumed a baseline GPU that does not perform address translation prior to accessing L1 data cache.

Therefore, extending TLB and page tables to support access classification is not very useful for such GPUs.

To enable access classification prior to accessing L1 cache, we add a memory access classifier (MAC) at each memory partition that keeps track of how sectors are shared among SMs. These classifiers are statically partitioned; all requests for a given address go to a single classifier. A sector is said to be unsafe if it is being accessed by multiple SMs and at least one of them is writing to it. Sector level classification is also cached at each SM in a Type-cache to quickly determine SM-safe or unsafe nature of memory accesses.

Preserving SC during a sector's transition to unsafe state: A sector starts in one of safe states and may eventually transition to the unsafe state (labeled as $\langle shared, rw \rangle$, Figure 5.6). During transition to unsafe state, a sector cannot immediately complete the transition because there could still be pending unsafe accesses that precede already completed safe accesses to this sector in program order. In order to satisfy SC ordering constraints, these unsafe accesses must be completed before transition to the unsafe state is completed. During this transition, all new requests to this sector are delayed till the sector has completed its state transition.

Preserving SC for SM-safe accesses: In SM level classification of accesses, it is possible that an SM-safe location is not "safe" at thread level. Therefore, freely reordering SM-safe accesses could result in an SC violation if multiple threads are performing conflicting accesses to SM-safe locations. Consider our earlier example from Figure 5.2b and assume that all locations are in global memory with X, Y as unsafe and Z as SM-safe. Since accesses to Z can get completed before preceding unsafe accesses, an SC violation occurs when $T3$ receives old value of X after reading latest value of Z because unsafe accesses from different threads get completed independently. This violation also occurs when Z is in shared memory because accesses to global and shared memory are not ordered with each other.

To guarantee SC execution, we need to identify when threads are performing conflicting accesses to an SM-safe location and delay conflicting accesses such that preceding unsafe accesses have been completed. In our previous example, we delay the read of Z in $T2$ till unsafe accesses in $T1$ prior to $I2$ (previous access of Z) are completed. To this end, we envision a small table (referred to as *synchronization table*) that keeps track of SM-safe accesses that have bypassed earlier unsafe accesses from same warp. This table is used to delay later conflicting accesses from getting performed until it is safe to do so.

Synchronization table only keeps track of SM-safe accesses that have gotten reordered with preceding unsafe accesses. We do not need to consider earlier SM-safe accesses (e.g. if X is classified as SM-safe) because in the proposed design all threads have same path to memory hierarchy (either to L1 cache or to shared memory banks). Due to this common path, threads running on an SM cannot observe reordering of two SM-safe accesses from another thread running on the same SM. Furthermore, if a thread from another SM attempts to access a SM-safe location, it will trigger state transition for the sector; resulting in completion of pending SM-safe accesses to this sector, and thereby preserving SC.

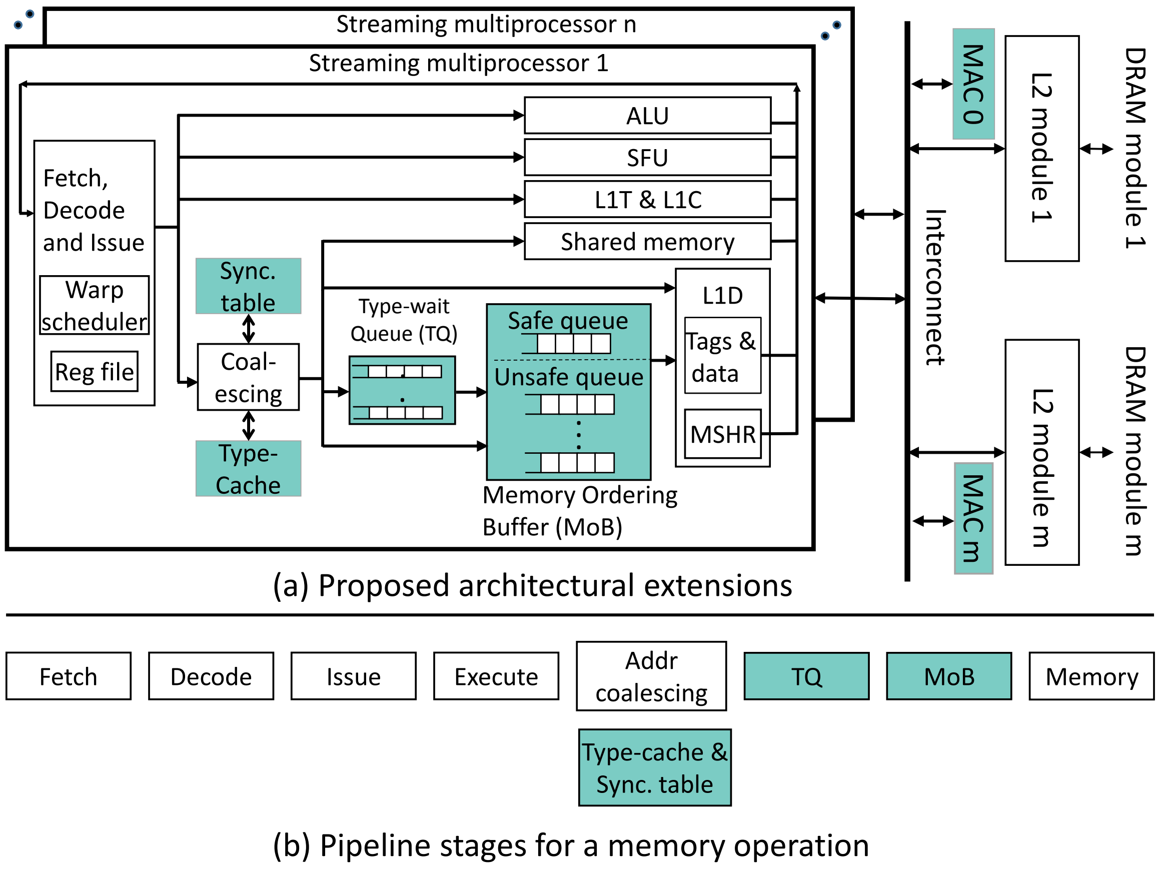


Figure 5.5: (a) Optimized SC GPU design with proposed changes highlighted in blue, (b) Pipeline stages for memory operations in proposed design.

5.4.2 Implementation

Figure 5.5a shows proposed extensions to a baseline GPU architecture. These changes include addition of Type-cache and memory access classifier (MAC) for classifying memory accesses, a synchronization table to ensure correct ordering for intra-SM conflicting accesses to an SM-safe location, and a memory ordering buffer to guarantee in-order commit of unsafe accesses. Figure 5.5b highlights how memory pipeline is extended to accommodate different functionalities.

5.4.2.1 Memory Access Classification:

As a first step, an SM needs to determine the type of a memory access prior to accessing L1 cache. Access type resolution cannot be carried out in parallel with the data cache access because this could lead to a SC violation. This can happen if the sector for this access needs to undergo a state transition. Such a transition has to happen before the cache access is done. Note that data cache is decoupled from access classification mechanism.

As mentioned earlier, we cannot rely on current GPU's virtual memory support for access classification. To facilitate access classification, we have extended each memory partition with a small memory access classifier (MAC). These classifiers keep track of SM level data sharing at sector granularity. Each entry in MAC stores identifier of previous accessor and state of the sector. An entry in MAC can be in one of following states: $\langle private, ro\rangle$, $\langle private, rw\rangle$, $\langle shared, ro\rangle$, and $\langle shared, rw\rangle$(Figure 5.6). In $\langle shared, ro\rangle$state, individual sharers are not tracked. Only the $\langle shared, rw\rangle$state

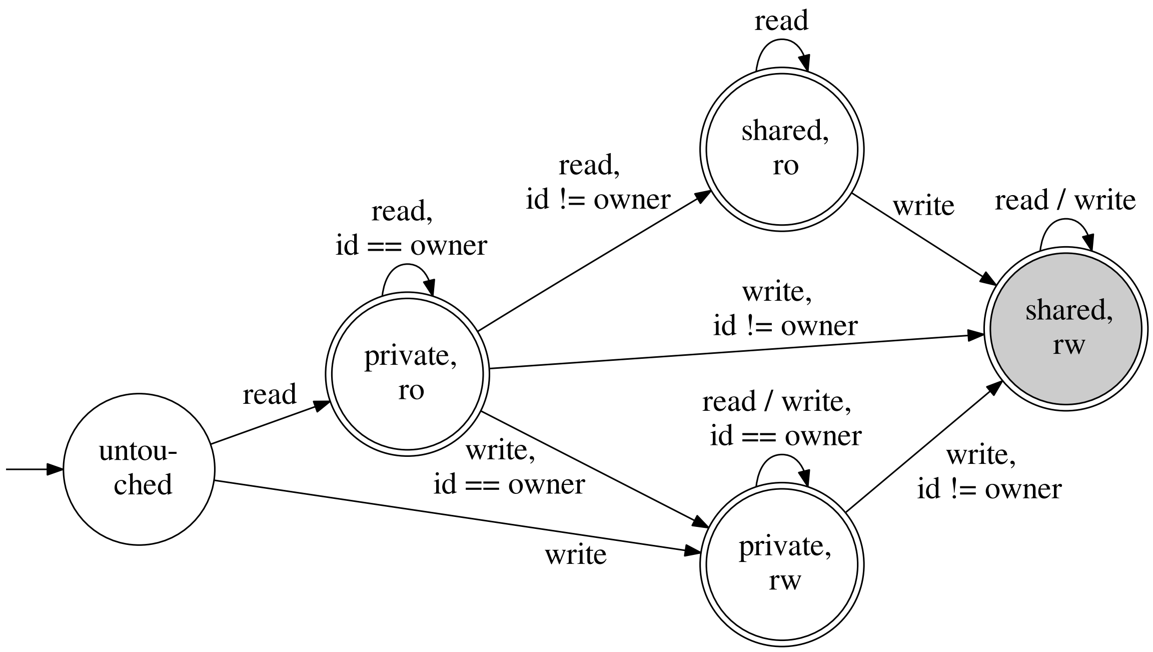


Figure 5.6: State transition diagram of sectors in MAC

is treated as unsafe state. This classification scheme is similar to schemes in prior proposals [48, 30, 104] that classified accesses at page granularity.

Accessing MAC incurs huge latency as requests have to cross interconnect. To make obtaining classification faster, we also cache a sector's information in a Type-cache in each SM. A Type-cache is analogous to TLB in CPUs. Besides a sector's classification, an entry in Type-cache also records an SM's write eligibility for this sector. If an SM does not have write privilege for a sector, it needs to send an request to MAC asking the same. These Type-caches are kept coherent through explicit messages on a sector's safe to unsafe state transition (*invalidation* message) and its eviction from MAC (*recall* message).

After a memory instruction is executed, addresses from different threads in a warp are coalesced before being sent to L1 cache to minimize L1 cache accesses. In parallel with address coalescing, an SM also accesses its Type-cache with address from a thread in the warp (Figure 5.5(b)). In the common case all threads are accessing only one cache block, their access type is already resolved without any additional delay. If multiple cache accesses are required for a memory instruction, their type resolution is also pipelined in the same way these accesses get sent to L1 cache. If an access misses in Type-cache, the SM sends a request to MAC to obtain the classification of the accessed sector.

On a miss in Type-cache, an access is moved to a type-wait queue (TQ stage) instead of blocking the pipeline. Type-wait queue is implemented as a per warp FIFO queue to avoid interference among different warps. FIFO queues ensure that accesses get their classification in program order. If a warp's type-wait queue is not empty, accesses from this warp are also sent to the type-wait queue even if they hit in Type-cache.

5.4.2.2 Ordering constraints for unsafe accesses

Once an access has been classified as SM-safe or unsafe, it can proceed with L1 cache access. Unsafe accesses are completed in their program order to guarantee SC. If an unsafe access cannot get issued to data cache because of ordering constraints, it is moved to unsafe FIFO queue for its warp in memory ordering buffer (MoB) to avoid pipeline stall (MoB pipeline stage). Once an access reaches at the head of its queue, it can be issued to cache.

Once an unsafe access has been issued to L1 cache, its entry is immediately reclaimed. To keep track of pending unsafe accesses, we can simply use per warp counters instead of keeping issued entries in unsafe queues. If a warp instruction accesses multiple cache blocks, these accesses can be issued to L1 cache in parallel because there is no ordering requirement on accesses from different threads running in a lock-step manner. However, unsafe accesses from later instructions are not allowed to access L1 cache till all preceding unsafe accesses have been completed.

5.4.2.3 Ordering constraints for SM-safe accesses

Ordering constraints for SM-safe accesses are relaxed except for conflicting accesses to an SM-safe location. We use *synchronization table* to identify when SM-safe accesses require strong ordering constraints. An SM-safe access at the head of type-wait queue searches synchronization table for conflicting accesses. If a match is found, this access needs to wait until the matching entry is deleted from the table. Otherwise, it can proceed with its L1 cache access. If an access is delayed due to a conflicting entry in the synchronization table, its warp is also restricted from issuing further memory requests till this wait is over.

Synchronization table is a small fully associative cache in each SM and supports a single writer or multiple readers for a cache block. An entry in this table has a pending count, access type (load or store) and warp identifier of previous accessor. An entry in the table is created when an SM-safe access bypasses pending unsafe accesses. An SM-safe access is said to have bypassed preceding unsafe accesses when it leaves the type-wait-queue and prior unsafe accesses in its warp are still pending. Furthermore, a marker entry is also inserted in the unsafe queue of this warp to determine when these unsafe accesses have been completed. This marker is deleted from the unsafe queue when all earlier pending unsafe accesses are completed. At this time, corresponding entry from synchronization table is also deleted because subsequent conflicting accesses do not need to wait anymore.

Synchronization table is also used to enforce ordering constraints for conflicting accesses in shared memory because shared memory accesses are akin to SM-safe accesses. Similar to SM-safe accesses in global memory space, shared memory accesses also search this table for conflicting entry and get delayed accordingly if a match is found. In our baseline GPU, shared memory accesses do not go through address coalescing stage. However, in our proposed design we also send shared memory accesses through an address coalescing stage to get 128 byte level unique addresses which are then used to index in the synchronization table. If a shared memory access needs to wait for a conflicting entry to get deleted from the table, its warp is also prevented from issuing further memory accesses to preserve intra-thread data dependencies.

Unlike for SM-safe accesses in global memory, additional markers are not inserted in an unsafe queue for shared memory accesses to get apprised of completion of pending unsafe stores. This avoids filling up of the unsafe queue with marker entries and keeps it available for unsafe accesses. Instead, we record an identifier for the latest unsafe access from this warp. This identifier is unique across all type-wait queues and unsafe queue entries. Once this unsafe entry is completed, synchronization table entries with this identifier are deleted from the table. In the case of multiple readers, each entry records one identifier per warp that has accessed the location pointed by this entry.

MoB also includes a safe queue for SM-safe accesses that could not be issued to L1 cache in this cycle due to cache port unavailability. The cache port could be unavailable if an access from either memory pipeline or from an unsafe queue wins arbitration for cache port. In the proposed design the safe queue is modeled as a FIFO queue even though there is no ordering requirement. We opted FIFO buffer to keep complexity low for the safe queue.

A memory operation can bypass TQ or MoB pipeline stages, if it hits in the Type-cache and there are no preceding pending unsafe accesses in its warp.

5.4.2.4 Preserving SC execution state on transition from safe to unsafe

When MAC detects a transition to the $\langle shared, rw\rangle$ state, it sends an *invalidation* message to the SM that had previously accessed this sector (or a broadcast if the sector was in $\langle shared, ro\rangle$ state). On receiving an *invalidation* request from MAC, an SM updates its Type-cache entry and other entries to this sector in its type-wait queue. It also needs to complete outstanding SM-safe accesses in the safe queue and drain its unsafe queues to ensure that all unsafe accesses that were bypassed by an SM-safe access to the sector indicated in the invalidation message have been completed. Once these queues have been drained, the SM sends an acknowledgment to MAC.

Draining all unsafe queues in MoB naïvely could be fairly expensive as there can many pending unsafe accesses from different warps that are not required to be completed in response to an invalidation message. In order to efficiently determine unsafe accesses that need to be completed, we reuse markers that are inserted in unsafe queues for SM-safe accesses. On receiving an *invalidation* request, an SM searches for markers to this sector in its unsafe queues. The SM needs to drain unsafe queues with a matching marker before sending an acknowledgment.

After receiving all acknowledgments (multiple acknowledgments for $\langle shared, ro\rangle$ state), MAC completes the safe to unsafe transition. To avoid races during this transition, all subsequent requests to this sector undergoing safe to unsafe state transition are not processed till this transition is completed.

5.5 Experimental Methodology

5.5.1 Simulation environment

We used GPGPU-sim [13] v3.2.1 to simulate the baseline NVIDIA Fermi architecture (GTX 480) and our proposed design for efficient SC. To simulate cache hierarchy of data caches we used Ruby memory systems from GEMS [78]. The baseline coherence protocol is MESI with write-allocate policy. The MESI coherence protocol we simulate is from gem5 [16]. We also evaluate a system configuration with write-through protocol with no-write allocate policy. This setup is based on the simulation infrastructure used in GPU-coherence [105] work.

Our baseline GPU system includes 16 multi-processors each with 32 KB of L1 data cache. The default warp scheduling policy is greedy-then-oldest-first (“GTO”) present in GPGPU-sim as it performs better than loose round-robin (“LRR”) policy. The optimized SC design uses GTO and gives higher priority to warps that do not have pending unsafe accesses.

We modeled a crossbar interconnect network using detailed fixed pipeline network model in Garnet [6]. Each SM is connected with the crossbar interconnect through private ports. We modeled minimum L2 and DRAM access latencies to be 340 and 460 cycles (in core cycles) respectively. These latencies are in accordance with latencies used in earlier work [105] and observed latencies on Fermi GPU via micro-benchmarks by Wong *et al.* [116]. Other relevant simulator parameters are listed in Table 5.2.

The proposed Type-cache and MAC are small caches and are connected through the interconnect. Messages for resolving an access' type can use the three existing virtual channels (request, forward, and reply) to achieve deadlock freedom.

Queues in type-wait queue and MoB hold both loads and stores. However, an store entry could be as large as a cache block size (128 bytes). To keep area requirements of these queues low, we limit the number of outstanding stores in these queues to 64 stores and keep the store data in a separate data array. This allows entries in queues to not provision for large store data. Since, there is no load-to-store forwarding, keeping store data in a separate buffer does not pose any additional challenge.

5.5.2 Benchmarks

To evaluate different SC/TSO designs, we studied applications from Rodinia [28] and Polybench [45] benchmarks suites, an GPU implementation of memcached [54]. Additionally, we also studied benchmarks used by Singh *et al.* [105] as these benchmarks exhibit a higher degree of inter thread block communication.

5.6 Experimental Results

In this section we compare the performance of TSO and SC (both naïve and optimized) against the baseline RMO memory model. We also show the impact of

Table 5.2: Simulator Configuration

GPGPU-sim Core Model	
# GPU Cores	16
Core Config	48 Wavefronts/core, 32 threads/wavefront, 1.4Ghz, Pipeline width:32, #Reg: 32768 Scheduling: Loose Round Robin. Shared Mem.: 48KB
Ruby Memory Model	
L1 Private Data$	32KB, 128B line, 4-way assoc. 128 MSHRs
L2 Shared Bank	128KB, 8-way, 128B line, 128 MSHRs. Minimum Latency: 340 cycles, 700 MHz
# Mem. Partitions	8
Interconnect	1 Crossbar/Direction. Flit: 32bytes Clock: 700 MHz. BW: 32 (Bytes/Cycle). (175GB/s/Direction)
# Virtual Networks	MESI: 5
GDDR Clock	1400 Mhz
Memory Channel BW	8 (Bytes/Cycle) (175GB/s peak). Minimum Latency: 460 cycles
Type-aware SC parameters	
Type-cache	64 entries, 4 way associative
Memory access classifier (MAC)	512 entries, 8 way associative, sector size: 16384 Bytes
Type-wait queue	48 FIFO queues with 4 entry per queue
Memory ordering buffer (MoB)	Safe queue: 16 entry FIFO; Unsafe queues: 48 FIFO queues with 4 entry per queue
Outstanding Stores in queues	64
Synchronization Table	64 entries

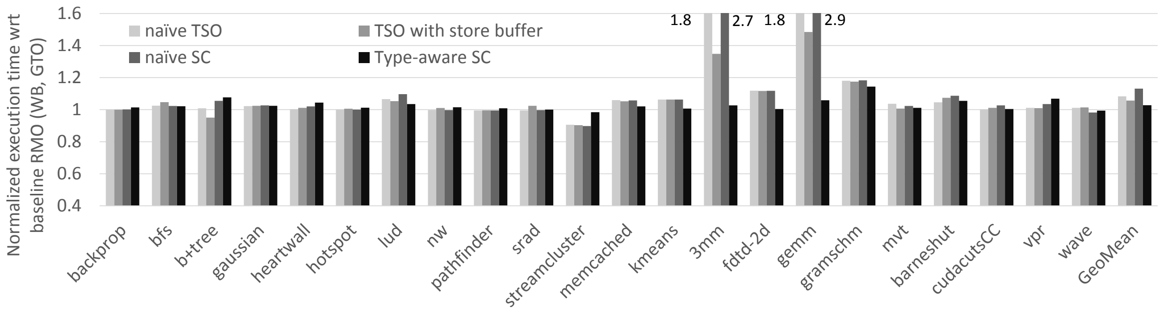


Figure 5.7: Normalized execution time of naïve TSO and naïve SC(write-back cache, GTO warp scheduler)

.

write-policy, warp scheduling policy on observed overhead of a given memory model.

5.6.1 Comparison of naïve SC, naïve TSO and RMO

Figure 5.7 shows normalized execution time of naïve SC and TSO designs with respect to a baseline GPU that implements RMO memory model. This graph reveals that for a majority of applications, naïve SC or TSO designs perform as good as baseline. There are two main reasons for this low performance gap between RMO and SC: warp-level-parallelism (WLP) and in-order execution. WLP helps in reducing the overhead of memory ordering constraints by allowing an SM to continue execution with different warps. Furthermore, the in-order execution could also limit RMO's ability to exploit intra-thread MLP (Section 5.3.4).

Although a majority of applications exhibit small performance difference for stronger memory models, there are benchmarks that incur huge performance overheads (up to 1.84x under TSO and up to 2.93x under SC for `gemm`). The primary sources of this performance gap is RMO's ability to take advantage of intra-thread MLP to overcome high cache miss rates. If a program has fairly low cache miss rates or lacks

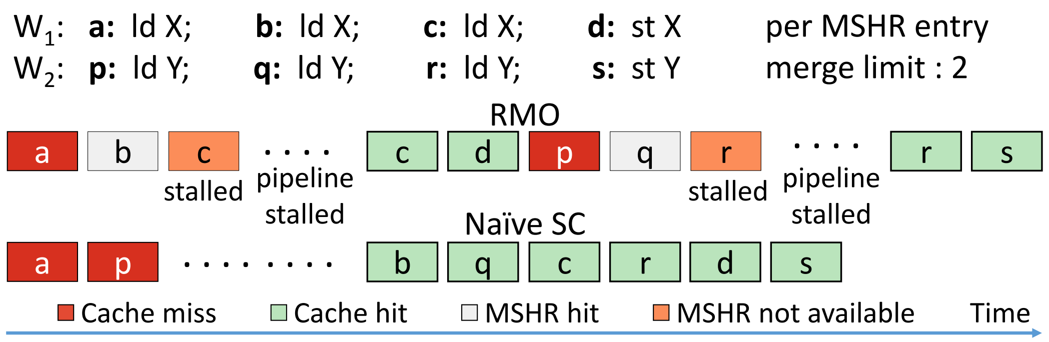

W1: a: ld X; b: ld X; c: ld X; d: st X
W2: p: ld Y; q: ld Y; r: ld Y; s: st Y
per MSHR entry merge limit : 2
RMO
a b c stalled .... pipeline stalled c d p q r stalled .... pipeline stalled r s
Naïve SC
a p .... b q c r d s
Cache miss
Cache hit
MSHR hit
MSHR not available
Time

Figure 5.8: An example to demonstrate how naïve SC could perform better than RMO

intra-thread MLP, SC can perform on par with RMO. For example, if we restrict each SM to execute only one thread block, these two benchmarks exhibit low cache miss rates (about 3%) which result in much smaller performance gap between SC and RMO. When an SM is allowed to execute up to 8 thread blocks, cache miss rates increase drastically (about 35%) due to higher contention at L1 cache. Under high cache miss rates, RMO performs better by exploiting intra-thread MLP in addition to WLP. Whereas, naïve SC is limited to exploiting just WLP.

Intuitively we expect RMO to outperform SC because it places fewer restrictions on execution. However, this may not always be true for GPGPU programs because there are many threads that could cause contention in memory hierarchy. For example, naïve SC and TSO outperform RMO for `streamcluster` (Figure 5.7).

To understand why SC outperforms RMO, consider an execution of code from `streamcluster` (Figure 5.8), where two warps W_1 and W_2 execute three independent loads and one store. Assume that all these warps access two distinct cache blocks that are not present in L1 cache. The first load from W_1 misses in the L1 cache and allocates an entry in MSHR. In RMO baseline, the GTO warp scheduler executes the second load from W_1 that hits in MSHR and gets merged with the existing entry in

MSHR. If each MSHR entry can merge only two memory requests, the third load from W_1 gets stalled because it cannot be merged into the existing MSHR entry, resulting in a stall in the memory pipeline. At this time, the SM cannot issue any new memory access until this stall is resolved even if it switches to other warps (e.g. W_2). Once the first cache miss satisfied, the SM can issue the third load and complete it. However, it runs into the same problem again when the third load from W_2 is not able to get merged into a existing MSHR entry.

In contrast, under SC the warp scheduler switches to warp W_2 after the load from W_1 has missed in the cache. However, after issuing one load from each warp, execution is stalled till these caches misses gets satisfied. In this case, the execution is stalled due to memory ordering constraints instead of resource contention. Once these loads have been completed, all subsequent loads accesses hit in the cache and get completed without any delay.

This example demonstrates how unrestricted issue of memory accesses can cause RMO to perform worse than SC (`streamcluster`). When we increase the limit on number of accesses that can be merged into an MSHR entry from 32 to 1024, RMO and SC have almost same performance. This is also the reason why our optimized Type-aware SC too performs worse than naïve SC.

5.6.2 Benefits of TSO over SC are Small

TSO enables store buffer optimization by allowing loads to bypass preceding stores. A store buffer can be organized either as an unified buffer or as a set per warp buffers. We find that unified store-buffer optimization performs significantly

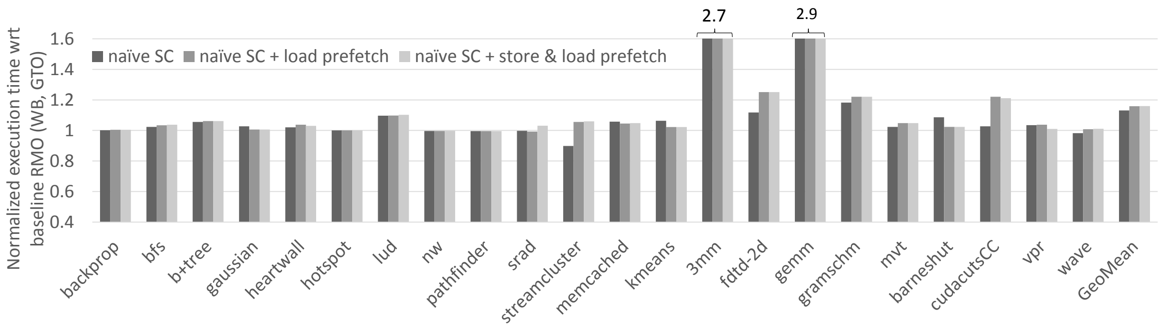


Figure 5.9: Normalized execution time of SC with prefetching (write-back cache, GTO warp scheduler)

.

worse than per warp configuration due to contention from different warps. Furthermore, Figure 5.7 shows that adding a large per warp store buffer (48 queues with 8 entries each with 128 bytes data) improves naïve TSO's performance only marginally.

For a majority of benchmarks both naïve SC and naïve TSO perform as good as RMO. Though TSO with store buffer performs better than naïve SC for other benchmarks, its overhead is still quite high in comparison to RMO. Marginal performance benefits of TSO for a few programs does not justify weakening SC guarantees.

5.6.3 Prefetching is Ineffective

In Section 5.3.5 we described an extension to naïve SC to take advantage of prefetching for accesses that cannot be issued to cache due to ordering constraints. Figure 5.9 shows the result for such a design. We find that prefetching helps very little in improving SC's performance. Whereas, some benchmarks (e.g., `streamcluster`, `fdtd-2d`) incur a performance loss due to prefetches being premature.

In CPUs, prefetches are used with in-window speculation to reduce memory ordering overhead. In window-speculation allows loads to consume prefetched data even if ordering constraints are not satisfied yet. However, in GPUs loads must still wait for ordering constraints to be satisfied due to lack of in-window speculation support. Furthermore, the probability of a prefetched data getting evicted before being read is also higher in GPUs due to large number of threads running on an SM.

Premature prefetches do not provide much benefit; at the same time they still consume limited resources like MSHR entries, space in L1 and L2 caches and bandwidth between L1 and L2 caches. Therefore, premature prefetches leads to poorer performance.

5.6.4 Type-aware design is Effective

Now, we evaluate performance of access type aware design presented in Section 5.4. Figure 5.7 shows normalized execution time of Type-aware SC with respect to a RMO baseline. From this graph, we find that Type-aware SC successfully brings down large naïve SC overheads in `3mm` and `gemm`. We also observe that the proposed design is not only able to reduce overheads of SC, but on average it performs very close to RMO. The primary source of this performance improvement is relaxing ordering constraints for SM-safe accesses.

Although Type-aware SC typically improves performance over naïve SC, it could also end up hurting performance. For example, for `b+tree` benchmark Type-aware design performs worse than a naïve design. The primary reason for this slowdown is unsafe queues being too small. A full unsafe queue can lead to memory pipeline stall if type-wait queue also become full. Increasing the size of unsafe queues alleviates this problem and, Type-aware SC performs very close to RMO. In case of `streamcluster`,

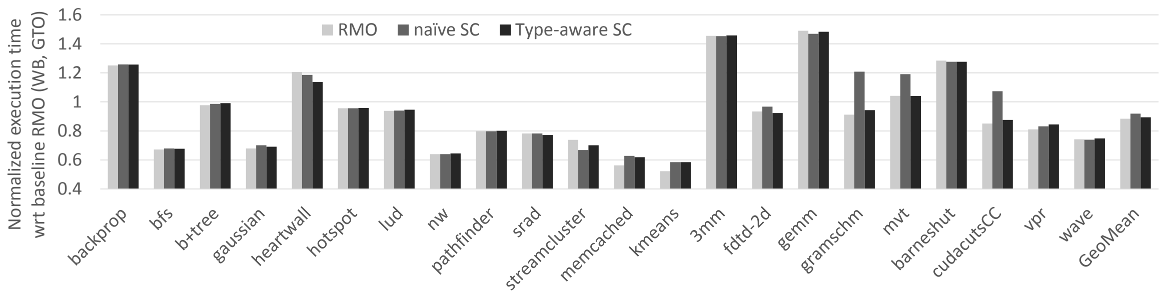


Figure 5.10: Performance of various memory models with a write-through L1 cache (GTO warp scheduler)

Type-aware SC also suffers from the same resource contention in MSHR that is present for baseline RMO.

Our sensitivity studies (not shown here) show that increasing the size of type wait queues and unsafe queues to more than four entries per warp provides very little performance improvement. Similarly, MAC and Type-cache also benefit very little from larger size due to bigger sectors and small shared L2 cache. The shared L2 cache in GPUs is small (less than 1 MB) in comparison to CPUs with even small core count. A smaller L2 cache results in a smaller set of locations that need to be tracked in MAC at any time.

5.6.5 Impact of cache-write policy

Current GPUs support write-through L1 data caches. To understand how a cache write policy affects the cost of memory ordering constraints, we evaluate performance of different memory models on a cache coherent GPU system with write-through and write no-allocate L1 caches.

Figure 5.10 shows performance impact of tradeoffs for write-through cache policy which are described earlier in Section 5.3.6. All execution times have been normalized to a system with RMO memory model and write-back caches. For the baseline RMO memory model, applications with streaming stores enjoy performance boost due to increased effective cache capacity for loads. However, some applications with non-streaming stores end up loosing performance due to increased contention in memory hierarchy (e.g., `gemm`).

The write-no allocate policy increases effective cache capacity because stores do not take any space in the cache. Both the baseline RMO and the naïve SC benefit from this increase in cache capacity and incur fewer load misses in comparison to write-back caches.

Intuitively, we expect performance gap between RMO and SC to be wider in write-through caches than the gap in write-back caches because stores have larger latency in write-through caches [90]. However, Figure 5.10 shows that this gap becomes narrower for write-through caches, e.g. naïve SC performs very close to RMO for `gemm` and `heartwall` benchmarks. Whereas, for write-back L1 cache, naïve SC has much higher overhead for these applications (Figure 5.7).

In applications with non-streaming stores, naïve SC still issues memory accesses in a restricted manner and typically experiences very small contention in memory hierarchy. Whereas, RMO could suffer from higher contention in memory hierarchy caused by additional stores being sent to L2 cache. This is indeed the case for `gemm` where RMO performs worse with write-through cache than with write-back cache. Due to these reasons there is very little difference in performance of RMO and SC for `gemm`. On the other hand, applications like `mvt` are hurt by higher store latency and exhibit larger difference between RMO and naïve SC under write-through caches.

For applications with streaming stores, sending stores to L2 cache on a write does not necessarily result in a performance loss because most of these stores will miss in L1 cache and get sent to L2 cache even with write-back L1 caches. On the other hand, both RMO and SC enjoy an increase in effective cache capacity. Therefore, relative performance difference between RMO and SC remains similar to that under write-back L1 caches.

5.7 Conclusion

In this work, we describe how stronger memory models (e.g. SC, TSO) can be implemented in GPUs. Despite GPUs executing instructions in order, they can violate memory ordering constraints by allowing out-of-order completion of memory accesses in a cache coherent system. We show that the inherent thread level parallelism prevalent in GPGPU programs helps close the performance gap between various memory models for many if not all benchmarks. We demonstrate using experimental evaluation, benchmarks which exhibit considerable performance overhead for stronger memory models and thus motivate the need to design optimizations which will help obviate this overhead. Furthermore, we also demonstrated that TSO memory model may not be an attractive alternative for GPUs as it does not provide significant benefits over SC. We evaluate efficacy of prefetching in bridging the performance gap of stronger memory models and find it to be insufficient. Finally, we adapt a non-speculative technique proposed for multi-cores to GPU architectures and show that it successfully eliminates the performance overhead for stronger memory models.

CHAPTER VI

Related Work

6.1 Efficiently Supporting Sequential Consistency

SC at language level can be provided by either compiler or a combination of SC-preserving compiler and hardware. In this section we discuss earlier works on supporting SC efficiently.

6.1.1 Compiler based approaches

In their seminal work Shasha and Snir [103] proposed "delay set analysis", which finds the minimum number of fences required for an SC execution. A fence incurs significant performance penalty on current processors. To optimize this cost, Sura et al. [107] and Kamil et al. [65] used static analyses to identify shared accesses and insert fences only for these accesses. More recently, Lin et al. [71] proposed `conditional fence`. They employ hardware support to enforce a fence ordering only when there is a possibility that SC may be violated. These static approaches use some form of whole program analysis that are hard to scale to large programs.

6.1.2 Data-Race Freedom by Construction

Since DRF-0 based memory models provide SC for only data race free programs, in order to provide SC, researchers have proposed to reject the racy programs during compile time. Several static type systems have been proposed for this purpose (e.g., [23, 18, 39]). While type systems provide a useful discipline on programmers to ensure race-freedom, they typically only account for lock-based synchronization and will reject race-free programs that use other synchronization mechanisms. Further, many correct programs that employ locks will be conservatively rejected due to imprecise information about pointer aliasing. More precision in static race detection can be achieved through inter-procedural analysis [94], but such whole-program analyses that tend to be heavyweight. In spite of recent advances [23, 22], a scalable and practically feasible technique for implementing a sound static data race detector also remains an unsolved problem, as all the techniques require complex, whole-program analysis.

6.1.3 Efficient SC hardware

In past several proposals have been made for efficiently supporting SC in hardware. Techniques used by these proposals can be classified in two categories: in-window speculation and out-of-window speculation. In-window speculation technique [41] allows loads in execution pipeline to execute speculatively even if there are earlier pending memory operations present in the pipeline. To detect a misspeculation, processor core snoops on cache invalidation requests. If there is no invalidation request to the cache block accessed by a speculative load, speculation is successful and load can be completed when it reaches at the head of ROB. In case of a misspeculation, processor pipeline is flushed and load is re-executed. Similarly for stores, eager read-exclusive prefetch requests can be issued as soon as store's address is known. Eager prefetch requests reduce the latency of store misses. In-window speculation can also be used to relax load-load ordering constraint in a TSO processor. Hardware support required for this technique is similar to that required for recovering from branch misprediction and ability to snoop on cache invalidations. This technique is relatively simple and modern processors already implement this technique [118, 60].

Second class of proposals use out-of-window speculation based techniques. With in-window speculation, a SC processor is able to relax memory ordering constraints for memory operation in pipeline. However, it must wait for earlier stores to get completed before it can commit next memory operation. To reduce this overhead, these proposals allow the processor to speculatively commit instructions if there are still stores pending [97, 44, 43, 46, 26, 114, 17]. To recover from misspeculation, processor needs to checkpoint not only register state, but also memory state. Processor snoops on coherence invalidation requests to detect a misspeculation. The checkpoint and rollback support required for this technique is fairly complex hardware support as it has to keep track of instructions even after they haven been committed from ROB. To avoid speculation, Lin et al. [72] proposed to check if there is any conflict with pending accesses in remote cores before committing a memory instruction from the ROB. While this design eliminates the need for out-of-window checkpoint and rollback support, it still requires significant changes to the coherence protocol to efficiently perform conflict detection before committing a memory instruction from the ROB.

In comparison to out-of-window speculation, SC hardware presented in Chapter IV avoids speculation by relaxing memory ordering constraints only when it is safe to do so without requiring a rollback of the execution. It uses a hybrid classification scheme to identify safe accesses and relaxes memory ordering constraint only for them. This design, therefore, does not require any complex checkpoint and rollback support. Splitting of store-buffer is almost straight forward as discussed in Chapter IV.

6.2 End-to-end Sequential Consistency

6.2.1 Transactional Memory

Hammond et al. [47] proposed transactional coherency and consistency (TCC) memory model based on a transactional programming model [53]. The programmer and the compiler ensure that every instruction is part of some transaction. The runtime guarantees serializability of transactions, which in turn guarantees SC at the language level. Unlike this approach, DRFx is useful for any multi-threaded program written using common synchronization operations like locks, and it does not require additional programmer effort to construct regions. TCC also requires unbounded region and speculation support. TCC suggests that hardware could break large regions into smaller regions, but that could violate SC at the language level.

Lazy conflict detection algorithm in DRFx is similar to the one proposed by Hammond et al. [47] but without the need for speculation and conflict detection over unbounded regions. Also, DRFx employs signatures to reduce the cost of conflict checks. Unlike TM, DRFx cannot afford false conflicts, which proposed DRFx design takes care to eliminate. But lazy conflict detectors like TCC assume some form of a commit arbiter to regulate concurrent commit requests for regions in different processors. As discussed earlier, DRFx can allow all regions to be conflict checked in parallel with the execution of current regions, which could be simpler. Also, soft-fenced regions can be executed and committed out-of-order.

6.2.2 BulkSC

BulkCompiler [7] and BulkSC [26] also provide end-to-end SC, but unlike programmer specified transactions, the BulkCompiler automatically partitions a program into regions called "chunks". These region-based solutions provide SC, but rely on fairly expensive speculation hardware (checkpointing, versioning, conflict detection, and recovery) to guarantee serializability of regions.

DRFx hardware could be simpler than Bulk hardware as it avoids the need for speculation (especially across I/O) and unbounded region sizes which have been the two main issues in realizing a practical transactional memory system. However, DRFx requires precise conflict detection, whereas Bulk can afford false conflicts. Our observations that certain regions can execute and commit out-of-order, and that conflict checks and region execution in different processors can all proceed in parallel is unique. It may help improve the efficiency and complexity of Bulk system as well.

SC-preserving compiler [76] and hardware (Chapter IV) together provide language level SC guarantees. SC-preserving hardware is much simpler than a HTM or BulkSC hardware because it does not require any complex checkpointing and rollback support, conflict detection mechanism found in these solutions.

6.3 Memory Models With Exceptions

The C++ memory model [20] and the Java memory model [74] are based on DRF-0 [3] and share its limitations for racy programs which we discussed in Chapter II.

Concurrently with work on DRFx, Lucia *et al.* [73] defined *conflict exceptions*, which also use a notion of regions to detect language-level SC violations in hardware. Their approach can be viewed as a realization of DRFx-compliant hardware, but it differs in important ways from our design. First, in their approach a conflict exception is reported *precisely*, just before the second of the conflicting operations is to be executed.

Precise conflict detection is arguably complex in hardware as one has to track access state for each cache word and continue to track it even when a cache block migrates to a different processor core. Further, when a region commits, its access state needs to be cleared in remote processors. Finally, while this approach delivers a precise exception with respect to the binary, the exception is not guaranteed to be precise with respect to the original source program.

Second, in their approach region boundaries are placed only around synchronization operations, thereby ensuring serializability of *maximal synchronization-free regions*, which is a stronger guarantee than SC. While this property could be useful for programmers, it can result in unbounded-size regions and thereby considerably complicates the hardware detection scheme and system software.

Adve *et al.* [5] proposed to detect data races at runtime using hardware support. Elmas *et al.* [36] augment the Java virtual machine to dynamically detect bytecode-level data races and raise a `DataRaceException`. Boehm *et al.* [19] provided an informal argument for integrating an efficient always-on data-race detector to extend the DRF-0 model by throwing an exception on a data race. However, detecting data races either incurs 8x or more performance overhead in software [38] or incurs significant hardware complexity [95, 84]. A full data-race detector is inherently complex as it has to dynamically build the *happens-before* graph [68] to determine racy memory accesses. It is further complicated by the fact that racy accesses could be executed arbitrarily "far" away from each other in time, which implies the need for performing conflict detection across events like cache evictions, context switches, etc. In contrast, DRFx hardware is inherently simpler as it requires that we track memory access state and perform conflict detection over only the uncommitted, bounded regions.

Gharachorloo and Gibbons [40] observed that it suffices to detect SC violations directly rather than data races. Their goal was to detect potential violations of SC due to a data-race and report that to the programmer. However, their detection was with respect to the compiled version of a program. DRFx incorporates the notion of compiler-constructed regions and allows the compiler and hardware to optimize within regions while still allowing us to dynamically detect potential SC violations at the language level.

More recently, Muzahid *et al.* [83] and Qian *et al.* [96] proposed hardware solutions to detect SC violation at the runtime. Instead of relying on data-races as proxy SC violations, they record some metadata about memory accesses, that complete before all earlier accesses have been completed, and pass on this metadata to other processors with coherence messages to detect cyclic dependence chains. These solutions detect SC violations with respect to execution of a given binary and do not account for reorderings performed by the compiler. They are, however, inadequate for detecting conflicts among regions as region serializability is strictly stronger property than sequential consistency. In absence of the notion of regions, it is not clear how these solutions would account for memory reordering done by compiler. These solutions, therefore, are not sufficient for guaranteeing language-level SC.

6.4 Private and Shared Data Driven Architectures

Trachsel *et al.* [110] proposed selective fences to optimize the overhead of fences in a RMO processor. In their proposal, a fence only needs to wait for stores to shared data (heap locations) get drained from store buffer. To keep track of stores to shared data, they proposed to split the store buffer into two separate store buffers. Our SC-hardware's proposal to split store buffer for tracking of shared stores is similar to selective fences, but our proposal provides more detailed design discussion on problems arising from having two store buffers (e.g. store-to-load forwarding, a escaped stack variable, etc.). SC-hardware also uses a more sophisticated classification scheme to identify safe accesses.

Past work has leveraged the page-protection mechanism for improving data placement in a processor cache [48], reducing snoops in a token-based coherence protocol [66], detecting thread dependencies to support replay [35], and more recently to improve the efficiency of directory caches [30]. Unlike these solutions, the goal of my research is to relax memory model constraints, which requires carefully orchestration of the state transitions of a page to ensure that memory ordering constraints are not violated. SC-hardware also employs a complementary static analysis technique to classify memory accesses.

6.5 Memory Consistency Models for GPUs

For throughput oriented processors with in-order cores, Hechtman *et al.* [52] observe that thread-level parallelism can mask the stalls due to strict memory ordering constraints. While this observation is valid for many GPU applications, we report several applications with significant SC overhead. Also, we study the interplay between memory ordering constraints and features common in GPUs such as write-through caches and warp-scheduling policies. Furthermore, in contrast to MTTOP systems they study, presence of a shared memory in GPUs allows threads in a thread-block to synchronize efficiently without increasing memory contention. Their observations on increased frequency of synchronization operations and memory contention issues due to them are less applicable for GPGPU programs.

Finally, we observe the ineffectiveness of prefetching optimization in reducing SC overhead. We propose an alternative optimization for GPUs that exploit spatial locality and "GPU-core locality" (data private to a core).

CHAPTER VII

Conclusion

The memory model of a concurrent language defines what values a load instruction can return. Semantics as fundamental as this should have a clean definition that matches the intuition of programmers. While the benefits of language-level sequential consistency are well known, an efficient *and* practically feasible solution for SC hardware has remained elusive.

The research in this dissertation demonstrates a safety-first approach for memory consistency models, whereby SC execution guarantee is provided to all programs, preventing unsafe execution behaviors arising from the presence of data-races in weaker memory models. The DRFx memory model and SC-preserving hardware (both CPU and GPU) are different instantiations of this approach and differ in how they handle unsafe accesses. The DRFx memory model allows common compiler and hardware optimizations on unsafe accesses and uses a runtime system to catch potential SC violations arising from reordering of unsafe accesses. Whereas, SC-preserving hardware focuses on enforcing SC instead of detecting SC violations.

The DRFx memory model for concurrent programming languages gives programmers simple, strong guarantees for all programs. Like DRF-0 based memory models, DRFx guarantees that all executions of a race-free program will be sequentially consistent. However, while DRF-0 based memory models typically give weak or no guarantees for racy programs, DRFx guarantees that the execution of a racy program will also be sequentially consistent as long as a memory model exception is not thrown. DRFx capitalizes on the fact that sequentially-valid compiler optimizations preserve SC as long as they do not interact with concurrent accesses on other threads. Since performing precise data race detection is impractically slow in software and complex in hardware, DRFx allows the compiler to specify code regions in which optimizations were performed. The hardware can then efficiently target data race detection only at regions of code that execute concurrently. This allows DRFx-compliant compiler and hardware to cooperate, terminating executions of racy programs that may violate SC. The implementation and evaluation indicate that a high-performance implementation of DRFx is possible.

SC-preserving hardware (in CPU) exploits an important opportunity that has been overlooked in the past while designing SC hardware: memory model constraints need not be enforced on accesses to private locations and shared, read-only locations. By exploiting this observation, SC-preserving hardware incurs low-complexity cost and obviates the need for aggressive speculation to obtain high performance. It uses a combination of static analysis and the page protection mechanism to identify safe accesses and relax SC constraints on them. Apart from an additional unordered store buffer, there is very little hardware modification needed to support our design. The end result is promising: SC hardware is only 4.5% slower than TSO, and end-to-end

SC costs only about 7.5% when compared to the performance of a state-of-the-art compiler and TSO hardware.

Further research extends our safety-first approach to data-parallel accelerators (e.g. GPUs) that are significantly different from CPU multi-cores. It also reveals that TSO, a widely used memory model in CPUs, has only a little performance advantage over SC, thereby giving little incentive to adopt TSO over SC for these accelerators. To support SC for these architectures, an SC-preserving design specific to GPUs is proposed that incurs only small performance overhead (about 3%) over baseline RMO. This work is not limited to only GPUs, but can also be applied to other throughput oriented engines such as Intel MIC, etc. Future heterogeneous systems will provide seamless integration of CPU and GPU; SC-preserving solutions for CPU and GPU can be combined in a novel way to provide strong SC guarantees for a heterogeneous system.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A.-R. Adl-Tabatabai and T. Gross. Source-Level Debugging of Scalar Optimized Code. In *Proceeding of the '96 Conference on Programming Language Design and Implementation*, 1996. 74

[2] S. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin–Madison, 1993. 60

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996. 8, 143

[4] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proceeding of the 17th Annual International Symposium on Computer Architecture*, 1990. 3, 10, 20, 21, 63, 96, 102

[5] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proceeding of the 18th Annual International Symposium on Computer Architecture*, 1991. 14, 15, 22, 144

[6] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proceeding of the 2009 IEEE*

Symposium on Performance Analysis of Systems and Software, pages 33–42, 2009. 127

[7] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *Proceeding of the 42nd Annual International Symposium on Microarchitecture*, pages 133–144. ACM, 2009. 60, 142

[8] 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 *Proceeding of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 577–591, 2015. 96, 108

[9] AMD. AMD I/O Virtualization Technology (IOMMU) Specification, 2006. 117

[10] AMD. AMD-V Nested Paging. White paper. `http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx`, 2008. 85

[11] AMD. AMD Graphics Core Next. `http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, 2011. 101, 107

[12] N. Amit, M. B. Yehuda, and B.-A. Yassour. Strategies for mitigating the iotlb bottleneck. In *Workshop on the Interaction between Operationg Systems and Computer Architecture*, 2010. 117

[13] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Proceeding of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009. 127

[14] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *SIGMETRICS*, 1998. 88

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceeding of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. 53, 62, 88

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. 127

[17] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proceeding of the 36th Annual International Symposium on Computer Architecture*, pages 233–244. ACM, 2009. 60, 64, 140

[18] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceeding of the 25th annual ACM SIGPLAN conference on Object-Oriented Systems and applications*, 2009. 139

[19] H. J. Boehm. Simple thread semantics require race detection. In *FIT session at PLDI*, 2009. 14, 22, 144

[20] H. J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceeding of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 2008. 3, 4, 11, 21, 31, 143

[21] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *Proceeding of the '10 Conference on Programming Language Design and Implementation*, pages 255–268, New York, NY, USA, 2010. ACM. 24

[22] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceeding of the 17th annual ACM SIGPLAN conference on Object-Oriented Systems and applications*, 2002. 139

[23] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *Proceeding of the 16th annual ACM SIGPLAN conference on Object-Oriented Systems and applications*, 2001. 139

[24] P. Cenciarelli, A. Knapp, and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Proceeding of the 16th European Symposium on Programming*, pages 331–346, 2007. 14

[25] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. The case for system support for concurrency exceptions. In *USENIX HotPar*, 2009. 17

[26] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceeding of the 34th Annual International Symposium on Computer Architecture*, pages 278–289. ACM, 2007. 60, 64, 140, 142

[27] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceeding of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238. IEEE Computer Society, 2006. 35, 38, 53

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceeding of the 2009 Workload Characterization*, pages 44–54, 2009. 128

[29] Compaq Computer Corporation. Alpha 21264 Microprocessor Hardware Reference Manual. Technical report, 1999. 85

[30] 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 *Proceeding of the 38th Annual International Symposium on Computer Architecture*, 2011. 60, 62, 80, 117, 122, 146

[31] R. Danilak. System and method for hardware-based gpu paging to system memory, Nov. 24 2009. US Patent 7,623,134. 117

[32] J. deok Choi, M. Gupta, M. J. Serrano, V. C, and S. P. Midkiff. Stack Alloca-

tion and Synchronization Optimizations for Java using Escape Analysis. *ACM Trans. on Programming Languages and Systems*, 2003. 76

[33] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009. 18, 29

[34] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceeding of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986. 9, 102

[35] G. W. Dunlap, D. G. Lucchetti, M. Fetterman, and P. M. Chen. Execution Replay on Multiprocessor Virtual Machines. In *VEE*, 2008. 80, 85, 146

[36] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *Proceeding of the '07 Conference on Programming Language Design and Implementation*, 2007. 22, 144

[37] The FeS2 simulator. http://fes2.cs.uiuc.edu/. 19, 53, 86

[38] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceeding of the '09 Conference on Programming Language Design and Implementation*, 2009. 15, 23, 144

[39] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *Proceeding of the '00 Conference on Programming Language Design and Implementation*, pages 219–232, 2000. 139

[40] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *Proc. of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, 1991. 15, 17, 24, 144

[41] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceeding of the International Conference on Parallel Processing*, pages 355–364, 1991. 39, 60, 61, 63, 69, 87, 97, 98, 113, 139

[42] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceeding of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990. 9, 20

[43] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *PACT*, 2002. 60, 64, 140

[44] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP=RC? In *Proceeding of the 26th Annual International Symposium on Computer Architecture*, pages 162–171. IEEE Computer Society, 1999. 64, 140

[45] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Autotuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012. 128

[46] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency

(TCC). In *Proceeding of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13. ACM, 2004. 64, 140

[47] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceeding of the 31st Annual International Symposium on Computer Architecture*, 2004. 35, 141

[48] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceeding of the 36th Annual International Symposium on Computer Architecture*, pages 184–195. ACM, 2009. 60, 62, 68, 80, 117, 122, 145

[49] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, 2012. 18, 29

[50] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. 2010. 2

[51] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 189–200, Feb 2014. 101

[52] B. A. Hechtman and D. J. Sorin. Exploring memory consistency for massively-threaded throughput-oriented processors. pages 201–212. ACM, 2013. 96, 146

[53] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceeding of the 20th Annual International Symposium on Computer Architecture*, 1993. 18, 141

[54] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceeding of the 2012 IEEE Symposium on Performance Analysis of Systems and Software*, pages 88–98, April 2012. 128

[55] M. D. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer*, 31:28–34, 1998. 60, 111

[56] M. D. Hill, A. E. Condon, M. Plakal, and D. J. Sorin. A System-Level Specification Framework for I/O Architectures. 1999. 85

[57] 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 *Proceeding of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014. 96, 107

[58] Intel. Intel Virtualization Technology for Directed I/O Architecture Specification, 2006. 117

[59] Intel Corporation . Intel architecture instruction set extensions programming reference. *319433-012 edition*, Feb. 2012. 18, 29

[60] Intel® Corporation. Pentium® Pro Family Developer's Manual. 1996. 60, 63, 140

[61] Intel® Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. *Order Number: 248966-025*, 2011. 71

[62] ISO/IEC 9899:2011. Programming language C. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`, 2011. 96

[63] W. Jia, K. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proceeding of the 20th International Symposium on High-Performance Computer Architecture*, pages 272–283, 2014. 114

[64] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceeding of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–406, 2013. 114

[65] A. Kamil, J. Su, and K. Yelick. Making Sequential Consistency Practical in Titanium. In *Proceeding of the 2005 ACM/IEEE conference on Supercomputing*, pages 15–. IEEE Computer Society, 2005. 138

[66] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace Snooping: Filtering Snoops with Operating System Support. In *PACT*, 2010. 62, 146

[67] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. *AMD*, 2012. 100

[68] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 23, 144

[69] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Computer*, 28(9):690–691, Sept. 1979. 2

[70] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceeding of the 2004 International Symposium on Code Generation and Optimization*, 2004. 19, 31, 52

[71] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *Proceeding of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010. 138

[72] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *Proceeding of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 273–286. ACM, 2012. 64, 140

[73] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions. In *Proceeding of the 37th Annual International Symposium on Computer Architecture*, 2010. 35, 48, 143

[74] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceeding of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2005. 3, 10, 20, 143

[75] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. 2009. 24

[76] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *Proceeding of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. 6, 142

[77] D. L. Marino. Simplified semantics and debugging of concurrent programs via targeted race detection, 2011. Copyright - Copyright ProQuest, UMI Dissertations Publishing 2011; Last updated - 2015-08-22; First page - n/a. 6

[78] M. M. 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. *ACM SIGARCH Computer Architecture News*, pages 92–99, 2005. 127

[79] S. P. Midkiff, D. A. Padua, and R. Cytron. Compiling Programs with User Parallelism. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, 1990. 60

[80] P. Mills, J. Lindholm, B. Coon, G. Tarolli, and J. Burgess. Scheduler in multi-

threaded processor prioritizing instructions passing qualification rule, May 24 2011. US Patent 7,949,855. 114

[81] A. Munshi et al. The OpenCL specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009. 7, 95

[82] A. Munshi et al. The OpenCL specification. *Khronos OpenCL Working Group*, 2.0, 2013. 96, 103

[83] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 363–375, Washington, DC, USA, 2012. IEEE Computer Society. 144

[84] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: Signature-based Data Race Detection. In *Proceeding of the 36th Annual International Symposium on Computer Architecture*, 2009. 15, 23, 144

[85] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races using Replay Analysis. In *Proceeding of the '07 Conference on Programming Language Design and Implementation*, 2007. 6

[86] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006. 85

[87] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Retrieved September, 2015. 7, 95, 103

[88] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi, White Paper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009. 101

[89] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110. *whitepaper*, 2012. 107

[90] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors. In *Proceeding of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1996. 115, 136

[91] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. In *Proceeding of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 743–758, 2014. 117

[92] B. C. Pierce. *Types and programming languages*. MIT press, 2002. 4

[93] J. Power, M. Hill, and D. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *Proceeding of the 20th International Symposium on High-Performance Computer Architecture*, pages 568–578, 2014. 99, 116, 117

[94] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceeding of the '06 Conference on Programming Language Design and Implementation*, pages 320–331, 2006. 139

[95] M. Prvulovic and J. Torrelas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded codes. In *Proceeding of the 30th Annual International Symposium on Computer Architecture*, June 2003. 23, 144

[96] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. Volition: scalable and precise sequential consistency violation detection. In *Proceeding of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 535–548, New York, NY, USA, 2013. ACM. 144

[97] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceeding of the ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210. ACM, 1997. 60, 64, 140

[98] T. Rogers, M. O'Connor, and T. Aamodt. Cache-conscious wavefront scheduling. In *Proceeding of the 45th Annual International Symposium on Microarchitecture*, pages 72–83, 2012. 114

[99] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceeding of the 46th Annual International Symposium on Microarchitecture*, pages 99–110, 2013. 114

[100] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP*, 2001. 76

[101] K. Sen. Race directed random testing of concurrent programs. In *Proceeding of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, New York, NY, USA, 2008. ACM. 24

[102] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *Proceeding of the 21st International Symposium on High-Performance Computer Architecture*, pages 174–185, 2015. 114

[103] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, Apr. 1988. 60, 99, 116, 138

[104] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end Sequential Consistency. In *Proceeding of the 39th Annual International Symposium on Computer Architecture*, pages 524 –535, june 2012. 117, 122

[105] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt. Cache coherence for gpu architectures. In *Proceeding of the 19th International Symposium on High-Performance Computer Architecture*, pages 578–590, 2013. 101, 127, 128

[106] S. Steele. ARM GPUs: Now and in the Future. 2011. 101

[107] Z. Sura, X. Fang, C. Wong, S. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Pro-*

ceeding of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2005. 138

[108] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. 1

[109] P. Tong, S. Yeoh, K. Kranzusch, G. Lorensen, K. Woo, A. Kaul, C. Case, S. Gottschalk, and D. Ma. Dedicated mechanism for page mapping in a gpu, Jan. 31 2008. US Patent App. 11/689,485. 117

[110] O. Trachsel, C. v. Praun, and T. R. Gross. On the Effectiveness of Speculative and Selective Memory Fences. In *Proceeding of the 2006 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2006. 145

[111] W. Triebel, J. Bissell, and R. Booth. *Programming Itanium-based Systems*. 2001. 34

[112] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Proceeding of the 22nd European conference on Object-Oriented Programming*, pages 27–51, 2008. 14

[113] D. Weaver and T. Germond. *The SPARC architecture manual*. 1994. 8, 108

[114] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *Proceeding of the 34th Annual International Symposium on Computer Architecture*, pages 266–277. ACM, 2007. 60, 64, 140

[115] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE*

conference on Supercomputing, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM. 33

[116] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Proceeding of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, pages 235–246, 2010. 128

[117] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceeding of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM, 1995. 53, 62, 88

[118] K. Yeager. The MIPS R10000 superscalar microprocessor. *Micro, IEEE*, 16(2):28–41, 2002. 140