UNIVERSITY OF CALIFORNIA,
IRVINE

An Initial Study of Two Approaches to Eliminating Out-of-Thin-Air Results

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Peizhao Ou

Dissertation Committee:
Professor Brian Demsky, Chair
Professor Rainer Dömer
Professor Guoqing (Harry) Xu

2018

DEDICATION

To my family

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor, Professor Brian Demsky, who has been incredibly supportive and an integral part of my doctoral research. His involvement and guidance have fueled my progress in every step of the way, and I have learned a ton about what qualities a good researcher should have while working with him. Without his persistent help and patience, this dissertation would not have been possible.

I would like to thank Professor Rainer Dömer and Professor Guoqing Xu, who have served on my candidacy exam committee and defense committee, for their invaluable time, contributions, and helpful feedback to my research. I would also like to thank many other researchers and anonymous reviewers for their very helpful discussion and feedback, which have helped improve my research.

I would also like to thank my previous and current colleagues in the *Programming Language Research Group*, Jin Zhou, Yong hun Eom, Brian Norris, Byron Hawkins, Ali Younis, Bin Xu, Rahmadi Trimananda, Hamed Gorjiara, and Zachary Snyder, for their invaluable advice, support and assistance. Specifically, I would like to thank Brian Norris for his wonderful CDSChecker model checking tool, upon which part of my doctoral research has been built.

I would like to thank Shuang Feng and Huan Tang, for giving me the internship opportunity at Google and their invaluable advice, guidance and patience during my internship period.

I would like to thank all of the great friends who I have made during my stay at UCI, for their help and support, and also for making the life of graduate school less boring.

Finally, I would like to thank my girlfriend, my parents, and my sister for their tremendous support, understanding and love in every aspect of my life.

My doctoral research and the work presented in this dissertation have been partly supported by a Google Research Award and the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, CCF1319786, OAC-1740210, and CNS-1703598.

CURRICULUM VITAE

Peizhao Ou

EDUCATION

Doctor of Philosophy in Computer Engineering **2012-2018**
University of California, Irvine *Irvine, CA, USA*

Master of Science in Computer Engineering **2012-2014**
University of California, Irvine *Irvine, CA, USA*

Bachelor of Science in Software Engineering **2008-2012**
Shanghai Jiao Tong University *Shanghai, China*

RESEARCH EXPERIENCE

Graduate Student Research **2012–2018**
University of California, Irvine *Irvine, CA, USA*

TEACHING EXPERIENCE

Teaching Assistant **Spring 16 & Spring 17**
University of California, Irvine *Irvine, CA, USA*

PEER-REVIEWED CONFERENCE PUBLICATIONS

Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results **Nov 2018**
International Conference on Object-Oriented Programming, Systems, Languages, and Applications

Checking Concurrent Data Structures Under the C/C++11 Memory Model **Feb 2017**
Symposium on Principles and Practice of Parallel Programming

AutoMO: Automatic Inference of Memory Order Parameters for C/C++11 **Oct 2015**
International Conference on Object-Oriented Programming, Systems, Languages, and Applications

Automatic Parameter Recommendation for Practical API Usage **Jun 2012**
International Conference on Software Engineering

SOFTWARE

OOTA-free Compiler Framework `http://plrg.eecs.uci.edu/oota-html/`
An LLVM-based C/C++11 compiler framework that prohibits OOTA behavior by preserving syntactic dependencies or atomic load-store ordering.

CDSSpec `http://plrg.eecs.uci.edu/cdsspec/`
A specification checker framework for concurrent data structures under the C/C++11 memory model.

AutoMO `http://plrg.eecs.uci.edu/automo/`
A tool that can infer memory order parameters for C/C++11 concurrent programs such that the programs only exhibit SC behaviors for the given test cases.

ABSTRACT OF THE DISSERTATION

An Initial Study of Two Approaches to Eliminating Out-of-Thin-Air Results

By

Peizhao Ou

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2018

Professor Brian Demsky, Chair

Eliminating so-called "out-of-thin-air" (OOTA) results is an open problem in many existing programming language memory models including Java, C, and C++. OOTA behaviors are problematic in that they break both formal and informal modular reasoning about program behavior. In spite of many years of research efforts, defining memory model semantics that are easily understood, allow existing optimizations, and forbid OOTA results remains an open problem. This dissertation explores two solutions to this problem that forbid OOTA results. The first solution is targeted towards Java-like languages in which all memory operations may create OOTA results, and the second solution is targeted towards C/C++-like memory models in which racing operations are explicitly labeled as atomic operations. Our solutions provide a per-candidate execution criterion that makes it possible to examine a single execution and determine whether the memory model permits the execution. We implemented and evaluated both solutions in the LLVM compiler framework. Our results show that on an ARMv8 processor the first solution has an average overhead of 3.1% and a maximum overhead of 17.6% on the SPEC CPU2006 C/C++ benchmarks, and that the second solution has no overhead on average and a maximum overhead of 6.3% on 43 concurrent data structures. The results indicate that these approaches to eliminating out-of-thin-air behaviors deserve further consideration.

Chapter 1

Introduction

With the wide scale deployment of multi-core processors, software developers must write parallel software to leverage the benefits provided by additional cores. In the meantime, mainstream hardware architectures primarily target shared-memory multiprocessors, in which multiple cores share the same main memory and communicate with one another through reading from and writing to the shared memory[1]. In a simplistic and ideal world, reasoning the execution of multi-threaded programs would be fairly straightforward: there exists a global view of the shared memory; each thread executes the memory operations strictly in program order, and a write to the shared memory immediately updates the shared memory and becomes visible to all the threads while a read from the shared memory retrieves the most up-to-date value. This intuitive abstraction was first introduced as *sequential consistency* by Lamport (1979).

However, in order to gain better performance, modern hardware violates sequential consistency in many ways. For example, out-of-order execution in modern processors can break the abstraction that memory accesses are executed in program order; or the memory subsystem

[1]In this dissertation, we are somewhat relaxed in our exchangeable usage of load/read to refer to the action of retrieving the value from a shared memory location, and store/write to refer to the action of updating a shared memory location with a value.

(e.g., store buffer or cache, etc.) can break the abstraction that a write is immediately visible to all the threads. Thus, we need a contract between a specific architecture and its users to specify what value a load can obtain for a multi-threaded program written at the machine-code level. Such a contract is known as a hardware memory model. A memory model in general specifies the semantics of memory reads/writes in a multi-threaded environment, and a hardware memory model does so at the hardware (or machine code) level.

In addition to the underlying hardware, standard compiler optimizations can also violate sequential consistency because they can transform the source code in such a way that read-/write operations appear to be reordered (Marino et al., 2011). In the meantime, the correctness of certain multi-threaded programs written in high-level programming languages can rely on the non-existence of certain relaxed behaviors[2] (e.g., reordering of memory accesses). Thus, in order to make source code portable across different compiler frameworks and hardware platforms, researchers and practitioners have designed and proposed programming language memory models, which serve as contracts between language users and language implementers to specify the semantics of loads and stores in a multi-threaded program at the programming language level. Most programming languages, such as C/C++ and Java, guarantee sequential consistency for programs without data races (Adve and Hill, 1990).

However, this guarantee is fragile — a single data race voids the sequential consistency guarantee for the entire program. Indeed, programs with data races have undefined semantics under the C/C++11 memory model (Boehm and Adve, 2008; JTC, 2011; Becker, 2011; Batty et al., 2011), mostly due to the fact that data races can violate the assumptions made by compiler optimizations. For existing compilers, assigning meaningful semantics to racy programs is extremely complicated. The language semantics must capture behaviors that arise from both compiler and processor optimizations. While the C and C++ memory models

[2]In literature, in the context of shared-memory multi-core systems, relaxed/weak (memory) behavior usually refers to the executions that are not sequentially consistent, and relaxed/weak memory models usually refer to the memory models that allow relaxed/weak behavior.

do not even attempt to assign semantics to such programs, Java is intended to support the safe execution of untrusted code. Thus, Java must ensure safety for racy programs and the Java Memory Model (Manson et al., 2005; Shipilëv, 2016a,b) attempts to assign semantics to such programs. A similar situation exists in C and C++ when it comes to the support for low-level `atomic` operations. The weakest `atomic` operations, which are specified using the `memory_order_relaxed` memory order parameter, only guarantee coherence and cannot be used to implement synchronization.

Hence, in both situations for Java and C/C++ relaxed atomics, one of the design goals is to define reasonable and preferably easy-to-understand semantics that allow standard compiler optimizations and modern hardware optimizations. This, however, has been shown to be very difficult, mostly due to an outstanding problem — eliminating out-of-thin-air (OOTA) results, i.e., results that can be justified only by some circular reasoning. For example, in a concurrent execution, we may justify the store of value `42` to memory location `x` only because we assume the exact store would happen in the first place. Such OOTA results have been shown to be disastrous because they hinder both formal and informal modular reasoning of concurrent programs. In this dissertation, we present two approaches to tackling this problem, i.e., the dependency-preserving approach and the load-store-order-preserving approach. We implemented both approaches on top of the LLVM compiler framework and evaluated their runtime overheads on an ARMv8 multiprocessor[3]. The remainder of this dissertation is structured as follows:

- In the remainder of this chapter, we discuss the background of sequential consistency (Section 1.1), hardware memory models (Section 1.2), compiler optimizations (Section 1.3), and programming language memory models (Section 1.4) in order to give readers sufficient background on how the out-of-thin-air problem arises.
- In Chapter 2, we depict the out-of-thin-air problem that this dissertation focuses on,

[3]The study presented in this dissertation is based on Ou and Demsky (2018).

show the disastrous consequences of allowing OOTA results, and outline existing proposals for solving the problem and our contributions.

- In Chapter 3, we demonstrate our memory model extensions for both of our approaches.
- In Chapter 4, we discuss our approach to extending the LLVM compiler to preserve our dependency notion.
- In Chapter 5, we discuss our approach to extending the LLVM compiler to preserve load-store ordering.
- In Chapter 6, we evaluate both approaches and show their runtime overheads.
- In Chapter 7, we discuss related work.
- In Chapter 8, we conclude and outline potential future work.

1.1 The Sequentially Consistent Memory Model

Sequential consistency, or the sequentially consistent (SC[4]) memory model, provides an intuitive abstraction that a multi-threaded program is executed as if there exists a strict interleaving of memory accesses from each thread that respects program order. Figure 1.1 shows the execution of a simple example[5] under the SC memory model. Here, variables `x` and `y` are two global variables which have the initial value of `0`, and variables `r1` and `r2` are local variables (or registers) that are initialized to be `0`. Then there are two threads, Thread 1 and Thread 2: Thread 1 writes value `1` to variable `x` and then reads from variable

[4]We loosely use the term "SC" to refer to either "sequentially consistent" or "sequential consistency" in this dissertation.

[5]We use pseudocode snippets like this throughout this dissertation. It is important to note that these examples are written in a C-like language whose purpose is to show the effect of memory accesses in a multi-threaded environment, and that they can actually represent code written in assembly, C/C++, or Java, etc., depending on the context. Also, in later examples, we use lower-case letters `x`, `y`, `z`, etc., to represent shared global variables, and symbols `r1`, `r2`, `r3`, etc., to represent local variables or registers, unless otherwise stated.

y; Thread 2 symmetrically writes value 1 to variable y and then reads from variable x. Under the SC memory model, depending on the actual interleaving of the two threads, there exist three possibilities for the values that variables r1 and r2 can hold: (1) r1=r2=1; (2) r1=1 $\wedge$ r2=0; or (3) r1=0 $\wedge$ r2=1. We show a possible interleaving of an SC execution that yields the result of r1=r2=1 in Figure 1.2, in which the two stores from both threads are executed before both loads. Notably, no execution would generate the result of r1=r2=0 under the SC memory model since both threads must respect the program order and thus write value 1 to either variable x or y before any reads from variable x or y can be executed. The Dekker's mutual exclusion algorithm (Dijkstra, 2002) essentially relies on the non-existence of this behavior (i.e., r1=r2=0) to provide mutual exclusion.

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

Figure 1.1: With x=y=r1=r2=0 initially, r1=r2=0 is not allowed under the SC memory model. The Dekker's algorithm essentially relies on the non-existence of this behavior to provide mutual exclusion.

Steps	**States of Shared Memory and Local Variables**
Initially, no thread executes	x=y=r1=r2=0
Thread 1 executes line "x=1;"	x=1 $\wedge$ y=r1=r2=0
Thread 2 executes line "y=1;"	x=y=1 $\wedge$ r1=r2=0
Thread 1 executes line "r1=y;"	x=y=r1=1 $\wedge$ r2=0
Thread 2 executes line "r2=x;"	x=y=r1=r2=1

Figure 1.2: A possible interleaving of Thread 1 and Thread 2 in Figure 1.1 which yields the result of r1=r2=1 under the SC memory model.

Given its simplicity, there exist many analyses, tools, concurrent data structures, and algorithms that are designed based on the SC memory model. In fact, researchers have explored approaches that just provide sequential consistency (Marino et al., 2011; Singh et al., 2012) for memory accesses for the C/C++ programming language. However, these approaches may require special hardware support to achieve competitive performance because mainstream

hardware does not implement sequential consistency. We will illustrate this in Section 1.2.

1.2 Hardware Memory Models

While the SC memory model is intuitive, it is not implemented by mainstream processors (e.g., x86, ARM[6], Power[7], and Itanium) for the sake of performance. Instead, these processors have relaxed memory models that are strictly weaker than the SC memory model and generally allow different levels of reordering of memory accesses.

1.2.1 Relaxed Behaviors Exposed by Hardware

The x86 Multiprocessors

The x86 multiprocessors are designed to have a relatively strong hardware memory model, which is formalized as the *x86-TSO* memory model (Owens et al., 2009; Sewell et al., 2010). It disallows load-load, load-store, and store-store reordering, but allows store-load reordering, i.e., allowing a later load to be reordered up across a previous store (in program order) to a different memory location. Also, the x86-TSO memory model enforces that there exists a total order for stores (to all memory locations) that is consistent across all the threads, which is referred to as *total store ordering* (TSO).

We use Figure 1.1 as an example to show the relaxed behavior that x86 processors can expose. Here, if we write the code snippet in Figure 1.1 in x86 assembly and run it on an x86 processor, it is possible that we observe the result of `r1=r2=0`. In this non-SC execution,

[6]We use the term "ARM" to refer to both the older version of ARM — ARMv7, and the more recent version of ARM — ARMv8, unless otherwise stated.

[7]We use the term "Power" in this dissertation to refer the multiprocessor family that includes various IBM POWER and PowerPC implementations.

both threads appear to perform store-load reordering and let both loads execute before the previous stores and read the stale values. One notable hardware optimization called *store buffering* can result in this execution. This optimization would add a store buffer unit that is local to each thread[8]. Then a store would go to the store buffer unit rather than directly to the main memory. To ensure per-location coherence, a load would first check whether the store buffer contains a store to its location or not; if so, it can do a thread-local store-to-load forwarding or otherwise would need to fetch the value from the shared memory (or a higher-level cache). At some point of the execution, the stores in the store buffer would be propagated to other cores. Hence, in this example, if both stores to variable `x` and `y` execute and stay in each thread's store buffer, and before they are propagated to the other thread, both loads to the opposite variables may execute and only see the initial value of `0`. This example is also known as the *store buffering* example.

The ARM and Power Multiprocessors — More Relaxed Behavior

The ARM and Power architectures have similar (but not identical) relaxed memory models (Maranget et al., 2012; Sarkar et al., 2011, 2012; Flur et al., 2016; Pulte et al., 2018), which are weaker than the x86-TSO memory model and can expose more relaxed behavior. First of all, not surprisingly, similar to the x86-TSO memory model, the ARM and Power memory models allow the store buffering behavior (i.e., the store-load reordering). In addition, in the architectural design of ARM and Power multiprocessors, there exist other aspects/optimizations that can expose other types of reordering of memory accesses, e.g., aggressive out-of-order execution such as speculative execution, more complicated store buffering optimizations such as hierarchical store buffering, and complex memory subsystem, etc. Also, as opposed to the total store ordering property in the x86-TSO memory model, the ARM and

[8]Here we picture a context where the store buffer unit is local to each thread, which is consistent with the x86-TSO abstract machine. However, for processors other than x86, depending on the actual processor design, the store buffer unit could be local to each thread, shared between threads on the same SMT core, or some more complicated design, etc.

Power memory models allow stores to different memory locations to propagate to different threads in different orders (with caveats). We will show examples to illustrate the relaxed behavior that the ARM and Power memory models allow in the following.

Message Passing — Store-Store & Load-Load Reordering Consider the *message passing* example shown in Figure 1.3. Here, variables `data` and `flag` are two shared variables, and variables `r1` and `r2` are registers, and they all have the initial value of `0`. Thread 1 can be viewed as a producer thread which updates the value of variable `data` and then updates variable `flag` to `1` to indicate to Thread 2 that variable `data` is ready to be consumed. Thread 2 can be viewed as a consumer thread, which continuously reads the `flag` variable until it has a non-zero value; once Thread 2 observes that the value of variable `flag` becomes non-zero, it starts to read variable `data`, which presumably should observe the value of `1` by the design of this producer-consumer scheme. In real-world code, one can expect Thread 1 to be the thread that initializes the fields of a complex object (i.e., store "`data=1`") and then assigns the pointer to the initialized object to a variable (i.e., store "`flag=1`"), and Thread 2 to be the thread that continuously checks if the pointer to that object is non-null (i.e., "`r1=flag`") and starts to read the fields of that object (i.e., "`r2=data`") if the pointer is non-null. Thus, the correctness of this code scheme essentially relies on the assumption that when the load from variable `flag` reads non-zero values, the later load from variable `data` must see the effect of the store "`data=1`".

If we write this code directly in x86 assembly, we are guaranteed that we will only observe the result of `r1=r2=1` because the x86-TSO memory model does not allow store-store or load-load reordering. However, for the same example, the ARM and Power memory models allow the non-SC result of `r1=1` $\wedge$ `r2=0` because they allow load-load reordering as well as store-store reordering for different memory locations. For example, under the ARM and Power memory models, stores in a thread may not propagate to other threads in program

order. Thus in Thread 1, the store to variable `flag` may propagate to Thread 2 before the store to variable `data`, as a result of which Thread 2 may see the update to variable `flag` but miss the update to variable `data`. It appears to Thread 2 that the two stores in Thread 1 are reordered. In addition, the two loads in Thread 2 can also appear to be reordered even if both stores are propagated to Thread 2 in program order. A possible scenario is that the processor performs speculative reads. In Thread 2, before the earlier load from variable `flag` returns value `1`, the later load from variable `data` can speculatively execute and return a stale value (i.e., value `0`).

data = flag = r1 = r2 = 0; // Initially	
Thread 1	Thread 2
`data = 1;` `flag = 1;`	`while (!(r1 = flag));` `r2 = data;`

Figure 1.3: The *message passing* example. With `data=flag=r1=r2=0` initially, although the execution of `r1=1` $\wedge$ `r2=0` is not allowed by the x86-TSO memory model, it is allowed by the ARM and Power memory models.

Load Buffering — Load-Store Reordering Another notable reordering that is architecturally allowed on ARM and Power is the load-store reordering. Consider the example shown in Figure 1.4, which looks similar to the store buffering example except that both threads perform a load first and a store to the opposite location second. The non-SC result of `r1=r2=1` is architecturally allowed on ARM and Power[9]. Once again, similar to the message passing example, x86 processors would disallow this execution as they do not allow load-store reordering.

[9]According to Maranget et al. (2012), this result has been observed on ARM processors and has not been observed on Power processors in practice, but it is architecturally allowed for both.

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2
`r1 = x;` `y = 1;`	`r2 = y` `x = 1;`

Figure 1.4: The *load buffering* example. With `x=y=r1=r2=0` initially, although the execution of `r1=r2=1` is not allowed by the x86-TSO memory model, it is architecturally allowed for ARM and Power.

1.2.2 Constraining Relaxed Behavior with Memory Fences and Dependencies

Although mainstream hardware can expose relaxed behaviors to users, they usually offer options to constrain relaxed behavior by enforcing stronger ordering guarantees. One important mechanism is to provide explicit machine instructions called *memory fences* (or *memory barriers*), or fences. The core idea is that given a specific fence instruction, certain types of memory operations that are before the fence (in program order) must become globally visible[10] before certain types of memory operations that are after the fence (in program order). For example, the `mfence` instruction in the x86 architecture requires that all memory operations before it (in program order) must be globally visible before any memory operations after it (in program order)[11]. Figure 1.5 shows a concrete example in which we use the `mfence` instruction to forbid the execution of `r1=r2=0` in the store buffering example, by adding an `mfence` instruction between each store-load pair in both threads.

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2

`x = 1;`
`mfence;`
`r1 = y;`
	`y = 1;`
	`mfence;`
	`r2 = x;`

Figure 1.5: With `x=y=r1=r2=0` initially, `r1=r2=0` is not allowed by the x86-TSO memory model because we insert an `mfence` instruction between every store-load pair in both threads.

[10] A load operation is considered to be globally visible when the value to be loaded into the destination register is determined.

[11] In the abstract machine of the x86-TSO memory model, the `mfence` instruction will flush the store buffer of that thread and propagate the stores to other threads.

Similarly, in the ARM and Power memory models, there exist similar fence instructions. The notable difference is that since the ARM and Power processors generally allow more reordering of memory accesses than the x86 architecture does, they provide more flexible fence variants. For example, the ARM architecture provides the full fence named "`dmb sy`" (or "`dmb`" for short), which is effectively similar to `mfence` in the x86 architecture (Maranget et al., 2012; Pulte et al., 2018); also, it provides weaker variants, such as the "`dmb ld`" instruction which only waits for loads (before the fence in program order) to complete, or the "`dmb st`" instruction which only waits for stores (before the fence in program order) to complete (ARM, 2016). Similarly, the Power architecture also provides different variants of fence instructions such as the `sync` instruction (also known as the heavyweight sync) which enforces store-store, store-load, load-load, and load-store ordering, and the `lwsync` instruction (also known as the lightweight sync) which is cheaper to execute than the `sync` instruction and enforces similar ordering constraints except store-load ordering (Sarkar et al., 2011; Maranget et al., 2012). For example, if we replace the `mfence` instructions in Figure 1.5 with either `dmb` or `sync` instructions in ARM or Power assembly, respectively, they can prohibit the execution of `r1=r2=0`, similar to using `mfence` instructions in x86 processors; however, fences like "`dmb ld`", "`dmb st`", or `lwsync` are not sufficient to prohibit the relaxed behavior in this case. In addition, the ARMv8 architecture introduces fence-like load/store instructions to enforce ordering constraints, which we will discuss in more details in Chapter 5.

In addition to fences, certain dependencies between hardware instructions can also constrain the reordering of memory operations. For example, Figure 1.6 shows a variant of the load buffering example, with the key difference that there exists a *control dependency* from the earlier load to the later store in both threads. Loosely speaking, under the ARM and Power memory models, when the result of a load is used to compute the condition of a conditional branch, and a store is after the conditional branch in program order, then the load forms a control dependency towards the store, which requires the load-store ordering to be respected (Sarkar et al., 2011; Maranget et al., 2012; Pulte et al., 2018). Hence, in

this case, the result of `r1=r2=1` is disallowed. There also exist other types of dependencies. For example, a *data dependency* roughly captures the dependency from the storing value of a store to the store itself, and an *address dependency* captures the dependency from the address of a load/store to the load/store itself. While dependencies can exist naturally in the source code as shown in Figure 1.6, one can also intentionally add dependencies to the assembly code for stronger ordering guarantees. For example, we can write the code shown in Figure 1.7 directly in ARM or Power assembly, in which the branch conditions are translated exactly as in "`r1==r1`" and "`r2==r2`". Then these intentionally added control dependencies can effectively forbid the load-store reordering in both threads and thus the result of `r1=r2=1` is disallowed.

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2

`r1 = x;`
`if (r1) {`
`  y = 1;`
`}`
	`r2 = y`
	`if (r2) {`
	`  x = 1;`
	`}`

Figure 1.6: A variant of the *load buffering* example. With `x=y=r1=r2=0` initially, `r1=r2=1` is not allowed by the ARM or Power memory models because in both threads there exists a control dependency from the earlier load to the later store.

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2

`r1 = x;`
`if (r1==r1) {`
`  y = 1;`
`}`
	`r2 = y`
	`if (r2==r2) {`
	`  x = 1;`
	`}`

Figure 1.7: The *load buffering* example with intentionally added control dependencies. Here, one should expect that this is ARM or Power assembly code with the branch conditions translated exactly as in "`r1==r1`" and "`r2==r2`". With `x=y=r1=r2=0` initially, `r1=r2=1` is not allowed by the ARM or Power memory models similar to Figure 1.6.

Note that hardware fences (depending on the actual type of fences) may incur significant runtime overhead, so developers generally use them only when a specific ordering constraint is necessary to guarantee the correctness of a multi-threaded program. For example, if we

want to implement the Dekker's mutual exclusion algorithm correctly on x86 processors, we must insert the `mfence` instruction appropriately similar to Figure 1.5.

It is important to note that the discussion about the x86-TSO, ARM, and Power memory models in this section is far from complete, but it should be sufficient for the purpose of illustration in the scope of this dissertation. Notably, the ARM and Power memory models would allow more relaxed behaviors such as allowing a store to be propagated to other threads at different time, etc., but mainstream hardware does enforce cache coherence — a per-location total order. Indeed, researchers have invested tremendous efforts in clarifying and formalizing hardware memory models vigorously. Notably, Owens et al. (2009); Sewell et al. (2010) formalize the x86-TSO memory model; Sarkar et al. (2011) formalize the Power memory model; Maranget et al. (2012) clarify the Power and ARMv7 memory model; and Flur et al. (2016); Pulte et al. (2018) formalize the ARMv8 memory model.

1.3 Compiler Optimizations

There exists yet another important aspect that attributes to exposing relaxed behaviors. Given a multi-threaded program written in a high-level programming language such as C/C++ or Java, etc., and suppose we have a specialized multiprocessor that only admits SC executions, we could still be subject to relaxed behaviors, due to the existence of compiler optimizations. Modern compilers have introduced many optimizations that help increase the performance of programs written in high-level programming languages, and there is a general rule behind these optimizations, usually referred to as the "*as-if*" rule, which roughly means that a compiler can transform a program as long as they do not change the observable behavior of the program. This, however, can expose relaxed behaviors in a multi-threaded environment.

Consider the example shown in Figure 1.8, which is a variant of the program shown in Figure 1.1. The difference is that this program is written in a high-level programming language such as C/C++, and that Thread 1 has one more load statement "`r0=y`" in the beginning. An optimizing compiler can transform this code such that the transformed program would exhibit the execution in which `r1=r2=0` even if it is run on an SC multiprocessor. The likely transformation done by the compiler follows: in Thread 1, the compiler realizes that the two loads from variable `y` are separated by a store that is to a different shared variable `x` (by points-to analysis the compiler can figure out that variable `x` and `y` point to different memory locations), and thus in a single-threaded environment, the second load from variable `y` is redundant (since the store "`x=1`" cannot change the value of variable `y`). Hence, instead of issuing two loads, the compiler can replace the second load simply with the result of the first one. Then the execution in which `r1=r2=0` can be produced by the following interleaving: (1) Thread 1 loads from variable `y` and `r0` obtains value `0`; (2) Thread 2 writes value `1` to variable `y`; (3) Thread 2 reads from variable `x` and `r2` obtains value `0`; (4) Thread 1 writes value `1` to variable `x`; and (5) variable `r1` reuses the result of variable `r0` (instead of reloading from variable `y`), which also obtains value `0`. This transformation also appears that the second load "`r1=y`" is reordered up across the store "`x=1`" and executes early. Note that there also exist other compiler optimizations that may appear to perform other types of reordering (Marino et al., 2011) .

1.4 Programming Language Memory Models

Given the fact that different multiprocessors and compilers can expose different levels of relaxed behaviors, it would be exceedingly difficult to write high-performance and portable multi-threaded programs. Moreover, Boehm (2005) argues that "threads cannot be implemented as a library", with the idea that a multi-threaded program may not be compiled

x = y = r1 = r2 = 0; // Initially	
Thread 1	Thread 2

`r0 = y;`
`x = 1;`
`r1 = y; // Redundant load`
	
	`y = 1;`
	`r2 = x;`

$\Downarrow Transformed$

Thread 1	Thread 2

`r0 = y; // Reads 0`
`x = 1;`
`r1 = r0; // Reuse earlier load`
	
	`y = 1;`
	`r2 = x;`

Figure 1.8: Given this program written in a high-level programming language such as C/C++, with `x=y=r0=r1=r2=0` initially, an optimizing compiler can transform the code such that the execution `r1=r2=0` is allowed even if the transformed program is run on a multiprocessor that only admits SC behaviors.

correctly (i.e., matching the users' intention) if a programming language is designed and implemented independently of threading issues. As a result, programming language designers and researchers have invested significant amount of effort in designing memory models at the programming language level. These programming language memory models serve as contracts between language users and language implementers and provide a layer of abstraction that is independent of specific compilers and hardware. In other words, if a developer writes a multi-threaded program based on a programming language memory model and uses a compiler that supports the memory model to compile the program (to a given ISA), the generated program should only exhibit behaviors allowed by the memory model.

To begin with, most programming languages provide sequential consistency for programs without data races (Adve and Hill, 1990; Gharachorloo et al., 1992) by default, usually referred to as the *SC-DRF* (sequential consistency for data-race-free programs) model. A data race in this context generally means "concurrently happening" memory accesses to the same memory location such that at least one of these memory accesses is a write. These programming languages provide specific concurrency primitives such as atomic read/write and mutex (e.g., lock/unlock) operations to protect the accesses to shared variables and guarantee se-

quential consistency if the program is free from data races. This is a nice property for many usage scenarios, i.e., programming with (default) concurrency primitives to ensure data race freedom; unfortunately, however, a single data race could break this guarantee. This leaves the following questions to the designers of programming language memory models: what semantics should we assign to programs with data races and to programs that require low-level support for atomic operations (e.g., in the C/C++ context). We will discuss the Java and C/C++ memory models to explore how these issues are addressed in different contexts in Section 1.4.1 and Section 1.4.2.

1.4.1 The Java Memory Model

The Java memory model (Manson et al., 2005; Shipilëv, 2016a,b) differentiates synchronization memory accesses from ordinary memory accesses by separating memory locations into two distinct parts: the `volatile` memory locations (variables declared with the `volatile` keyword) and ordinary (non-volatile) memory locations (any other non-volatile variables). Reads/writes to `volatile` locations are atomic accesses that are totally ordered (across different `volatile` locations) and can be used to establish synchronization between threads. In addition, conflicting `volatile` accesses are not considered data races in Java and thus can "happen concurrently" without voiding the sequential consistency guarantee.

However, when it comes to data races, the Java memory model becomes subtle. Java is designed to be a safe language, which means that it needs to provide at least some semantics even for incorrect (or untrusted) code, which includes code with data races. Precisely providing such semantics is extremely challenging because it should allow current (and potentially future) optimizations used in compilers and hardware while it should disallow those especially undesired relaxed behaviors. It is very important to note that though many efforts have been invested, the Java memory model is still not satisfactory, especially because it

disallows some common optimizations that JVMs actually perform, such as redundant read elimination (Cenciarelli et al., 2007; Ševčík and Aspinall, 2008).

1.4.2 The C/C++ Memory Model

Unlike Java, C/C++ is not a safe language and does not have the need to provide a safety guarantee for untrusted code. Hence, the C/C++ memory model (Boehm and Adve, 2008; JTC, 2011; Becker, 2011; Batty et al., 2011; Batty, 2014) designers have decided that C/C++ programs with data races would have undefined behaviors, which means it is the developers' responsibility to ensure that their program is free of data races. However, as an important programming language for many performance-critical systems, C/C++ does have the need to support low-level concurrency primitives used in scenarios such as lock-free algorithms. Similar to Java, to differentiate atomic memory accesses from ordinary memory accesses, C/C++ let users specify atomic objects and use atomic read, write, or read-modify-write operations to atomically read/write these objects. Also, operations on atomic objects are not considered data races. However, unlike the fact that Java only has `volatile` accesses, C/C++ atomic operations provide more flexibility for developers to make tradeoffs between the ordering guarantees provided and the overhead incurred, and they can have one of the six *memory orders*, each of which falls into one or more of the following categories:

seq-cst: `memory_order_seq_cst` – strongest memory ordering, there exists a total order of all operations with this memory ordering. Loads that are seq_cst either read from the last store in the seq_cst order or from some store that is not part of seq_cst total order. Note that `seq_cst` accesses essentially behave as `volatile` accesses does in Java.

release: `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a store-release may form release/consume or release/acquire synchronization. When a load-acquire reads from a store-release, it establishes a happens-before relation between

the store and the load.

acquire: `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a load-acquire may form release/acquire synchronization.

consume: `memory_order_consume` – a load-consume may form release/consume synchronization, which is only carried across to memory accesses that have dependencies on the load-consume (with caveats). It is important to note that researchers or practitioners have not found a clean and efficient way to implement load-consume yet, and that mainstream compilers (including the latest version of GCC and Clang/LLVM compilers) usually treat them as the relatively stronger load-acquire.

relaxed: `memory_order_relaxed` – weakest memory ordering. The main design goal of relaxed atomics is that one can exactly translate them into a plain load/store instruction in machine code without adding extra synchronization constraints. Hence, the only constraint for relaxed atomics is a per-location modification order total ordering (also known as *cache coherence*) that is equivalent to the per-location coherence property in hardware memory models. Therefore, relaxed atomics cannot be used to implement synchronization, and the effects of compiler optimizations can in some cases be visible to relaxed atomics. Thus in some sense relaxed atomics have qualitatively similar behaviors to ordinary memory accesses in Java.

The C/C++ memory model then expresses program behavior in the form of binary relations or orderings. We briefly summarize the relations:

- **Sequenced-Before:** The evaluation order within a program establishes an intra-thread *sequenced-before* (*sb*) relation—a strict preorder of the atomic operations over the execution of a single thread.

- **Reads-From:** The *reads-from* (rf) relation consists of store-load pairs (X, Y) such that Y takes its value from X—or $X \xrightarrow{rf} Y$. In the C/C++ memory model, this relation is non-trivial, as a given load operation may read from one of many potential stores in the execution.

- **Synchronizes-With:** The *synchronizes-with* (sw) relation captures the synchronization that occurs when certain atomic operations interact across threads.

- **Happens-Before:** In the absence of memory operations with the `consume` memory ordering, the *happens-before* relation is the transitive closure of the union of the sequenced-before and the synchronizes-with relations. Note that a data race is precisely defined as two memory accesses such that they are not ordered by the *happens-before* relationship and at least one of them is a non-atomic write.

- **Sequentially Consistent:** All operations that declare the `memory_order_seq_cst` memory order have a total ordering (sc) in the program execution.

- **Modification Order:** Each atomic object in a program has an associated *modification order* (mo)—a total order of all stores to that object—which informally represents an ordering in which those stores may be observed by the rest of the program.

Program executions directly observe the reads-from relation by observing the values that loads return. The synchronizes-with, happens-before, sequentially consistent, and modification order orderings constrain the reads-from relation and are only indirectly observable by the effect that they have on the reads-from relation. We show and discuss a few concrete examples written under the C++ memory model in the following.

Using the Strongest Ordering Parameter `seq_cst` Figure 1.9 shows a C++ version of the store buffering example shown in 1.1. Here, both variables `x` and `y` are declared as the `atomic_int` type rather than the normal `int` type, and all loads/stores are associated with

the `seq_cst` ordering parameters. We use Figure 1.10 to show why the non-SC execution of `r1=r2=0` is not allowed in this example. This graph shows the *sc* (i.e., the total order for `seq_cst` operations) and *rf* (i.e., *reads-from*) edges of the execution of `r1=0` $\wedge$ `r2=1`; the nodes represent memory operations, and the green edges represent the *sc* edges, and the red edges represent the actual *rf* edges. The prefix `Init` in the nodes indicates the nodes are from the initialization thread (before Thread 1 and Thread 2), and the prefix `T1` indicates a node is from Thread 1, and the prefix `T2` indicates a node is from Thread 2. Since the `seq_cst` memory operations form a total order, and loads must read from the latest store in that *sc* order, the load "`r2=x.load()`" in Thread 2 cannot read from the initial value. Similar reasoning applies to other possible *sc* total orders of the memory operations in this example, and hence the non-SC execution of `r1=r2=0` is disallowed.

atomic_int x, y;	
Thread 1	Thread 2
`x.store(1, seq_cst);` `int r1 = y.load(seq_cst);`	`y.store(1, seq_cst);` `int r2 = x.load(seq_cst);`

Figure 1.9: With `x` and `y` declared as atomic variables and all memory accesses to them associated with the `seq_cst` memory order parameter, this C++ program will only admit SC executions, i.e., `r1=r2=0` is not allowed.

Using the `release` & `acquire` Ordering Parameters for Synchronization Figure 1.11 shows a C++ version of the message passing example, in which variable `data` is a non-atomic variable and variable `flag` is an atomic variable, and the store to and load from the variable `flag` use the `release` and `acquire` ordering parameters, respectively. Figure 1.12 depicts the execution graph with *sb* (*sequenced-before*), *sw* (*synchronizes-with*), *hb* (*happens-before*) and *rf* edges for this example. As we can see, whenever the load-acquire from variable `flag` reads value `1` from the store-release to variable `flag`, they establish a *synchronizes-with* edge. Together with the *sequenced-before* edges in both threads, ordinary store "`data=1`" *happens before* ordinary load "`r2=data`", which ensures that there does not exist a data race for the load/store on variable `data` and that load "`r2=data`" must see

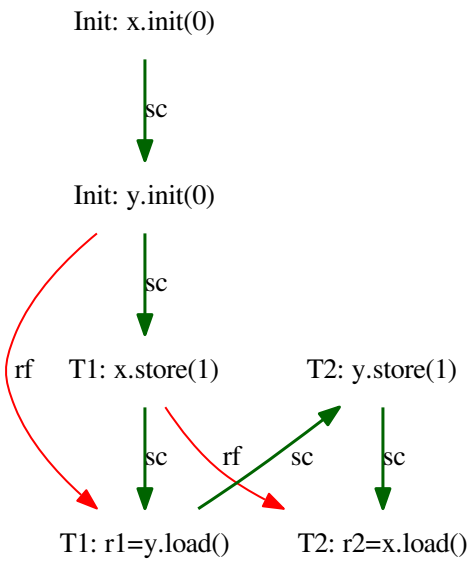

Figure 1.10: The execution graph for the example shown in Figure 1.9. The total order *sc* forbids the execution of both loads reading the initial stores.

the side effect of store "`data=1`". Note that if we replace the `release` or `acquire` ordering parameters with the `relaxed` ordering parameter, there would be data races in the program and the program would have undefined semantics according to the C++ memory model.

int data = 0; atomic_int flag;	
Thread 1	Thread 2
`data = 1;` `flag.store(1, release);`	`while (!(r1 = flag.load(acquire)));` `int r2 = data;`

Figure 1.11: With `data` declared as an non-atomic variable, `flag` declared as an atomic variable, `data=flag=0` initially, the `release` store and `acquire` load together establish a *synchronizes-with* edge and thus ensure that the ordinary store `data=1` *happens-before* the ordinary load `r2=data`, and thus this code is data-race-free and the execution of `r1=1` $\wedge$ `r2=0` is not allowed.

Using the Weakest Ordering Parameter `relaxed` Figure 1.13 shows a C++ version of the load buffering example, in which all shared variables are declared as `atomic_int`

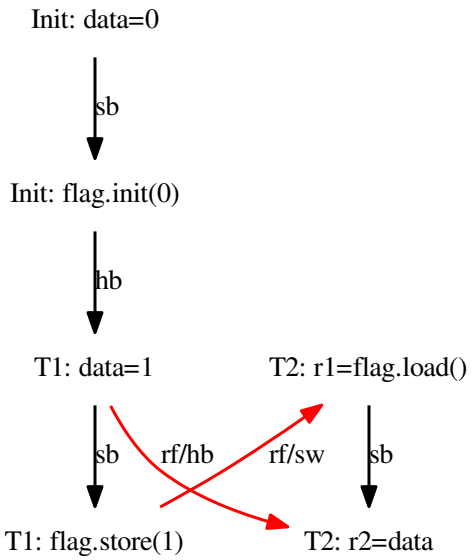

Figure 1.12: The execution graph for the example shown in Figure 1.11. The *sw* edge establishes *happens-before* relationship between the store/load on variable `data`.

and all memory operations use the `relaxed` ordering parameter. This program does not have a data race since all memory accesses are atomic operations; however, since `relaxed` atomics themselves do not establish synchronization, the execution of `r1=r2=1` is allowed. We show its execution graph in Figure 1.14, in which there exists a cycle in the union of *sequenced-before* and *reads-from* edges. As discussed above, `relaxed` atomics are designed to be subject to compiler and processor optimizations, and the example here shows that with `relaxed` atomics, the C/C++ memory model allows the load-buffering behavior.

atomic_int x, y;	
Thread 1	Thread 2
`int r1 = x.load(relaxed);` `y.store(1, relaxed);`	`int r2 = y.load(relaxed);` `x.store(1, relaxed);`

Figure 1.13: With `x` and `y` declared as atomic variables and all memory accesses to them associated with the `relaxed` memory order parameter, the C++ memory model allows the behavior of `r1=r2=1`.

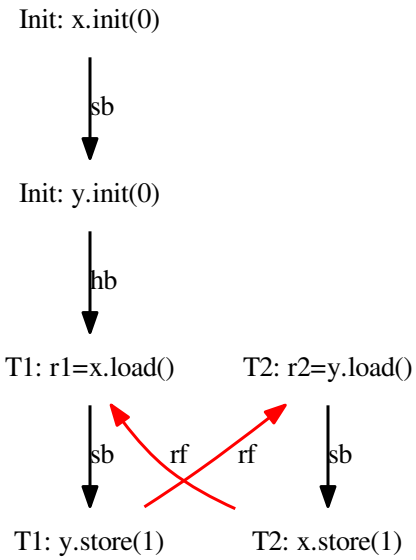

Figure 1.14: The execution graph for the example shown in Figure 1.13. The loads and stores with the `relaxed` ordering parameter do not establish synchronization, and the C/C++ memory model allows cycles in the union of the *sb* and *rf* relation.

Chapter 2

Overview of the OOTA Problem

Although the Java and C/C++ memory models have different design requirements and focuses, there exists a very important commonality: both Java's non-volatile memory accesses and C/C++'s relaxed atomics are designed to be subject to existing compiler and hardware optimizations. In both contexts, there still exists an outstanding open problem — eliminating *out-of-thin-air* (or OOTA) results. Indeed, neither the Java nor C/C++ memory models have a satisfactory solution to this challenging problem, even though the C/C++ memory model is generally thought to be the state-of-the-art design of programming language memory models. A significant amount of the complexity of the Java memory model comes from eliminating OOTA results, yet so far it yields a memory model that forbids some common compiler optimizations, and solving this problem remains exceedingly challenging. The C/C++ memory model on the other hand takes a "hand-wavy" solution, which does not clearly and satisfactorily define OOTA results and only vaguely states that implementations should not produce out-of-thin-air results.

In this dissertation, we focus on studying the runtime costs of two approaches that require strengthening existing memory models to eliminating out-of-thin-air behaviors, hoping that

our effort will help programming language memory model designers better understand the potential runtime costs of these solutions, which we believe are promising approaches that are worth further study and consideration. In this chapter, we will discuss the OOTA problem, its undesired consequences, and potential solutions in more details.

2.1 The Problem

A key challenge in programming language memory models is prohibiting out-of-thin-air behaviors or satisfaction cycles. This problem is well known (Batty et al., 2013; Pichon-Pharabod and Sewell, 2016; Manson et al., 2005; Batty et al., 2015a) and has been described in detail (Boehm and Demsky, 2014). Figure 2.1 presents an execution that real processors produce. As discussed in Section 1.2.1, a processor might reorder the store "`x=42`" up across the load "`r2=y`" in Thread 2, Thread 1 can then read the value `42` from `x` and store it in `y`, and finally Thread 2 can load `42` from `y`. If we write this code with C/C++ atomics and assign the `relaxed` ordering parameter to all loads and stores, the C/C++ memory model also allows this execution (i.e., `r1=r2=42`).

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = 42;`

Figure 2.1: "Real Example". With `x=y=0` initially, can `r1=r2=42`?

Figure 2.2 presents an out-of-thin-air example with the same reads-from relationship between loads and stores as the previous example, as shown in Figure 2.3. If both loads read from the subsequent stores, and all the loads/stores have the `relaxed` ordering parameter, the C and C++ memory model formalism admits the execution in which `r1=r2=42` (or any other value), conjuring the value of `42` "out of thin air". Notably, with only the execution graph shown in Figure 2.3, we are not able to distinguish the two examples by the C/C++ memory model. The key difference between these two examples is that in the problematic example,

the stores depend on the previous loads.

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Figure 2.2: Canonical Out-of-Thin-Air Example. With `x=y=0` initially, can `r1=r2=42`?

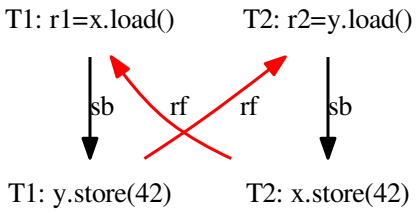

Figure 2.3: The execution graph for both executions in Figure 2.1 and Figure 2.2. Though the two examples are different, both executions are allowed by the C/C++ memory model formalism, and we cannot differentiate one from another by the *sequenced-before* and *reads-from* relationship.

Note that if we directly write these examples in assembly, no processor will produce the problematic results in Figure 2.2. Processors preserve a notion of dependency — a processor core will not make a speculative store visible to other cores. Compilers in general do not preserve dependencies — compiler optimizations can easily optimize away dependencies (e.g., an `if` statement in which both branches store the same value to the same variable). Compiler optimizations conspire with relaxed hardware implementations to create the problem.

Although there is agreement that Figure 2.2 represents OOTA behavior and should be prohibited, the precise definition of OOTA behavior is disputed. Consider the example (from Boehm and Demsky (2014)) shown in Figure 2.4. An optimizing compiler may discover that the load `r1=x` in Thread 1 will always return the value of `42` no matter whether the conditional branch is taken or not. Hence it can replace the store `y=r1` with `y=42`[1], and then through

[1]This optimization is implemented for non-atomic memory accesses in both the GCC and Clang/LLVM

the same reads-from relationship between loads and stores as Figure 2.2, the execution in which `r1=r2=42` is allowed. While some researchers argue that this is OOTA behavior and should be disallowed, other researchers may argue that this is legitimate behavior because the value of `42` in this example arises from the untaken branch.

Thread 1	Thread 2

`r3 = x;`
`if (r3 != 42)`
`  x = 42;`
`r1 = x;`
`y = r1;`
	`r2 = y;`
	`x = r2;`

Figure 2.4: With `x=y=0` initially, can `r1=r2=42`? While there seems to be general consensus that the execution shown in Figure 2.1 is not OOTA behavior and that the execution shown in Figure 2.2 is OOTA behavior, researchers may not agree whether the execution in this example is OOTA behavior.

2.2 Consequences

Historically, Java was the first general-purpose shared-memory programming language that attempts to assign semantics to concurrent code, with the two major design goals: (1) allowing existing compiler and processor optimizations and (2) providing sufficiently strong guarantees for concurrent code to function correctly. However, the original Java language specification (Gosling et al., 1996) was shown by Pugh (1999) to fail to achieve either goal. It was both too strong to admit common compiler optimizations and too weak to support some commonly used programming idioms. Various proposals for fixing the original model were found to either prohibit optimizations or allow undesired behaviors (Maessen et al., 2000; Manson and Pugh, 2001; Adve, 2004; Saraswat, 2004). Later, Manson et al. (2005) proposed a new Java memory model, which provides safety guarantees including disallowing out-of-thin-air executions. This is a major design requirement for a safe language like Java,

compilers. Although both compiler frameworks do not optimize this case for relaxed atomics at this point, it is a legitimate optimization (by the C/C++ memory model) which they may adopt in the future.

and one of the reasons is that out-of-thin-air values on references (e.g., suppose `r1` and `r2` are object references in Figure 2.2) would allow the program to do anything. Yet, again, this has been shown to unsound with respect to some common compiler optimizations that are actually performed by Java compilers (Ševčík and Aspinall, 2008; Cenciarelli et al., 2007), and this problem remains unresolved.

The C/C++11 memory model on the other hand has been shown to be sound with respect to existing compiler optimizations (Morisset et al., 2013), but unfortunately the memory model formalism allows out-of-thin-air executions, and the issue is also unresolved. As previously noted (Boehm and Demsky, 2014; Batty et al., 2013; Vafeiadis et al., 2015; Vafeiadis and Narayan, 2013), allowing out-of-thin-air results is disastrous. Serious issues of allowing OOTA results include:

1. **OOTA Results Break Formal Modular Reasoning:** As noted by Batty et al. (2013), OOTA executions can break certain types of compositional reasoning about programs. In particular, even if the guarantees provided by each component satisfy the assumptions of all other components, OOTA results allow executions in which two components mutually violate their own guarantees and thus violate the assumptions of the other component (circularly justifying their violation of their guarantees).

   In languages that allow OOTA results, compositionality requires proving that each component in a composition is non-interfering (i.e., that it does not write to the memory locations of other components). Indeed, some analyses and tools that are based on the C/C++ memory model either simply assume the non-existence of OOTA behavior or require a stronger version of the C/C++ memory model that prohibits OOTA behavior (Norris and Demsky, 2013; Ou and Demsky, 2015; Meshman et al., 2015; Kokologiannakis et al., 2017).

2. **OOTA Results Break Informal Modular Reasoning:** While developers rarely

formally prove their software correct, OOTA results can even break informal reasoning about programs. Indeed, for many programs it may be necessary to avoid including accesses to relaxed atomics in the code base. For example, simply exposing an interface to relaxed stores to a virtual machine interpreter is likely sufficient to allow OOTA results that can produce arbitrary executions.

3. **OOTA Results Can Affect Race-Free Computations:** OOTA results can induce race-free computations to produce surprising results. The following example is courtesy of Sarita Adve:

Thread 1	Thread 2
if (x) y=1;	if (y) x=1;

Even with `x=y=0` initially, OOTA results allow this computation to set both `x` and `y` to 1.

Figure 2.5 shows a concrete example borrowed from Boehm and Demsky (2014) that helps illustrate the severity of allowing OOTA behaviors. In this example, `Foo` is a struct with an atomic pointer pointing to the next `Foo` object, and variable `a` and `b` are two shared pointers pointing to two disjoint lists of `Foo` objects. If we allow OOTA behavior, the execution in which `r1=b` $\wedge$ `r2=a` would be allowed. The reason is that Thread 1 can assume the load from `a->next` returns value `b`, and then updates `b->next` with value `a`, and thus Thread 2 loads value `a` from `b->next`, and finally updates `a->next` with value `b`, which satisfies the assumption in the first step. This as a result links the two lists `a` and `b`, which presumably should be disjoint. It is clear that allowing such behavior is unacceptable as it breaks the reasoning of almost any code.

```
struct Foo {
 atomic<Foo*> next;
};
Foo *a, *b;
```

Thread 1	Thread 2
`Foo* r1 = a->next.load(relaxed);` `r1->next.store(a, relaxed);`	`Foo* r2 = b->next.load(relaxed);` `r2->next.store(b, relaxed);`

Figure 2.5: The ghostly linking of disjoint lists because of allowing OOTA behavior.

2.3 Potential Solutions

Researchers have proposed several different basic approaches to solving the out-of-thin-air problem. We next discuss the different basic approaches. These approaches fall into two primary categories. The first category attempts to eliminate the problem by changing the memory model specification without significant changes to the compiler and the second category attempts to eliminate the problem by providing stronger guarantees.

2.3.1 Approaches that Primarily Affect the Language Specification

Precisely Specifying the Effects of Existing Optimizations: Forbidding OOTA behaviors by precisely specifying the effects of existing optimizations is one potential solution. This approach is tempting as it incurs no runtime overheads and requires no modifications to either compilers or processors. Researchers have proposed event-structures-based memory models (Jeffrey and Riely, 2016; Pichon-Pharabod and Sewell, 2016) that were later shown not to be compiled to ARM without additional fences in some cases (Kang et al., 2017).

Attempts at forbidding OOTA executions by precisely specifying the effects of optimizations have to date yielded complicated memory models. Indeed, Batty et al. (2015b) show that there is no per-candidate-execution solution to the problem. For example, Kang et al. (2017)

propose a memory model based on a semantics that claims to resolve the OOTA problem, and the proof of its compilation correctness has been shown by Podkopaev et al. (2017). While this approach can potentially solve the OOTA problem, a less complex but slightly stronger memory model may still be desirable if the overhead is acceptably small.

The Java Memory Model also attempted this approach (Manson et al., 2005), but the approach has since shown to be unsound with respect to standard compiler optimizations (Ševčík and Aspinall, 2008; Cenciarelli et al., 2007). Moreover, the Java memory model is extremely complicated for both compiler developers and application developers to understand. It is also complicated to use the constraints placed on OOTA executions by the Java memory model to prove correctness properties for concurrent programs. Indeed, Botinčan et al. (2010) have shown that for the Java memory model, the problem of verifying causality requirements for a finite execution of an arbitrary multi-threaded program is undecidable.

Case-Based Approaches: Another approach is to constrain the usage of atomics to specific cases and then provide simple semantics for those cases. The most well known example of this approach is the classic "data race freedom implies sequential consistency" memory model used by most multi-threaded programming languages (Adve and Hill, 1990). In this model, if there is no sequentially consistent execution with a data race, then the system guarantees that all executions are sequentially consistent. Other work enumerates common use cases for relaxed atomics and provides semantics for those use cases (Sinclair et al., 2017).

There are two basic challenges with this approach: (1) memory model developers must ensure that the cases handled cover the important usage scenarios and (2) bugs can produce behaviors that fall outside the well defined cases and then the memory model may provide little or no guarantees as to the program's behaviors.

2.3.2 Approaches that Provide Stronger Guarantees

Approach 1: Forbid Load-Store Reordering: A conceptually simple approach to forbidding OOTA executions is to forbid load-store reordering (Boehm and Demsky, 2014; Lahav et al., 2017). Precisely, the memory model requires that **sequence-before**∪**reads-from** is acyclic. This greatly simplifies the memory model for both compiler and application developers, but can potentially incur significant runtime costs.

Implementing this approach requires changes to compiler optimizations and the potential generation of a fence-like operation. The cost of this approach depends on both the details of the memory model and the hardware architecture. While x86 processors already provide this behavior without requiring fences, architectures like ARM or PowerPC may incur higher overheads. We believe that this approach is likely to be acceptable for memory models like C/C++11 as it only affects relaxed loads and stores[2]. However, this approach affects all loads and stores in Java programs, and thus is likely to be less acceptable in the context of Java.

Approach 2: Preserve Dependencies: Earlier work suggested but did not implement one potential approach to forbidding OOTA executions — require the compiler to preserve a simple, syntactic notion of dependency (Boehm and Demsky, 2014). Effectively, this approach provides a syntactic definition for a **dependency** relationship and then requires that **dependency**∪**reads-from** is acyclic. This is a strictly weaker guarantee than the previous approach. It is worth noting that the Linux kernel memory model does not have out-of-thin-air values because it essentially respects syntactic dependencies as hardware does (Alglave et al., 2018). McKenney et al. (2016) have proposed an approach based on preserving semantic dependencies rather than syntactic dependencies. For example, they allow reducing

[2]Under C/C++11, non-atomic loads and stores cannot race, or the program has no semantics. Thus, reordering cannot be observed.

an expression with syntactic dependency to a constant if it is known to always result in that constant value (e.g., reducing "`r1=x*0`" to "`r1=0`"). This trades off the simplicity of the memory model specification for the degree of compiler optimizations that are allowed. In this dissertation, we explore an approach of preserving syntactic dependencies.

While this approach does not require the addition of any extra fence instructions, it does constrain the optimizations performed by the compiler. The primary concern with this approach is that the overheads were previously unknown and feared to be high.

2.4 Contributions

This dissertation makes the following contributions:

- **A dependency-based approach to forbidding OOTA:** It presents a dependency-preserving approach to forbid out-of-thin-air executions.
- **An approach to preserving load-store ordering to forbid OOTA:** It presents an approach to preserving load-store ordering to extend C/C++-like language memory models to forbid out-of-thin-air executions.
- **Implementations of both approaches in the LLVM compiler:** It presents implementations of both approaches to forbidding OOTA in the LLVM compiler.
- **Evaluation:** It evaluates the overhead of both approaches on an ARMv8 processor. It shows that the average overhead of preserving dependencies relative to compiling with full optimizations (-O3) is 3.1% on the SPEC CPU benchmarks for a prototype implementation that is likely amenable to further optimizations. It shows that under our experimental setting preserving load-store ordering has no overhead on average and worst-case overhead of 6.3% on concurrent data structure benchmarks.

Chapter 3

Memory Model Extensions that Disallow OOTA Behaviors

In this chapter, we first discuss a dependency-preserving memory model that disallows out-of-thin-air behaviors by defining a notion of dependency and preserving it. While it is desirable that we formalize this memory model in the context of an existing programming language, e.g., C/C++ or Java, both the C/C++ and Java languages are complex and the formalization would exceed the scope of this dissertation. Therefore, we introduce a simple language that captures the core features of an imperative programming language in Section 3.1. Then, we define the notion of dependency based on this language and describe how we preserve such dependencies in Section 3.2. Last, we discuss a load-store-order-preserving memory model that prevents out-of-thin-air behaviors in Section 3.3.

3.1 The Language

Figure 3.1 presents the core syntax of our language[1]. To simplify the illustration, we only support one value type — *numerals*. When we read from or write to a global variable/memory location, we explicitly use the **load** or **store** keywords to distinguish them from assignments to local variables. Our language is based on the static single assignment (SSA) form (Rosen et al., 1988), where we can have phi functions (ϕ) at the end of an `if/else` block or in the beginning of a `while` loop's header. The syntax starts with *Program*, which has an optional declaration of global variables followed by a list of function definitions. Figure 3.2 shows an example code snippet written in our language. In this example, we declare three global variables `x`, `y`, and `z` and perform load/store from/to these global locations in line 3, 6 and 10, respectively. Line 9 is the ϕ function for the `if/else` conditional branch, meaning that if the condition "`r2==0`" is `true`, local variable `r5` will be assigned with `r3`; otherwise, it will be assigned with `r4`.

3.2 Language-Level Dependency Notion

While there is general agreement about extreme examples that conjure new values and exhibit out-of-thin-air behavior (e.g., the example shown in Figure 2.2), there is no consensus on the exact definition of an out-of-thin-air execution. In this dissertation, we broadly define an out-of-thin-air execution to be any execution in which the behavior of an operation is circularly involved in causally justifying its own behavior. To prohibit such executions, we define a conservative syntax-based notion of dependency that maps loads in a thread to all stores in the thread whose behavior the loads may affect. We also provide a proof sketch of a theorem about the causality of executions in our memory model in Section 3.2.1. The precise

[1]The toy language here is purely for the purpose of simplifying illustration, and our actual implementation is for C/C++.

$$
\begin{array}{rcl}
\textit{Var} & ::= & \text{Variable Names} \\
\textit{Func} & ::= & \text{Function Names} \\
\textit{Num} & ::= & \text{Constant Numerals} \\
\textit{op}_b & ::= & \text{C-like binary operators} \\
\textit{op}_u & ::= & \text{C-like unary operators} \\
\textit{Expr} & ::= & \textit{Num} \mid \textit{Var} \mid \textit{FuncCall} \mid \textit{Expr}\ \textit{op}_b\ \textit{Expr} \\
 & & \mid \textit{op}_u \textit{Expr} \mid \textbf{load}\ \textit{Expr} \\
\textit{VarList} & ::= & \textit{Var}\ (\text{“,”}\ \textit{Var})^* \\
\textit{ExprList} & ::= & \textit{Expr}\ (\text{“,”}\ \textit{Expr})^* \\
\textit{Phi} & ::= & \textit{Var}\ \text{“=”}\ \text{“}\phi\text{”}\ \text{“(”}\ \textit{Var}\ \text{“,”}\ \textit{Var}\ \text{“)”} \\
\textit{PhiList} & ::= & (\textit{Phi}\ (\text{“;”}\ \textit{Phi})^*)? \\
\textit{Stmt} & ::= & \textbf{skip} \mid \textit{Stmt}\ \ \text{“;”}\ \ \textit{Stmt} \mid \textit{Var}\ \text{“=”}\ \textit{Expr} \mid \\
 & & \textbf{store}\ \textit{Expr}\ \text{“,”}\ \textit{Expr} \mid \textit{FuncCall} \mid \\
 & & \textbf{return}\ \textit{Expr}? \mid \\
 & & \textbf{if}\ \textit{Expr}\ \textbf{then}\ \textit{Stmt}\ \textbf{else}\ \textit{Stmt}\ \textbf{fi}\ \textit{PhiList} \mid \\
 & & \textbf{while}\ \textit{PhiList}\ \textit{Expr}\ \textbf{do}\ \textit{Stmt}\ \textbf{od} \\
\textit{FuncDef} & ::= & \textit{Func}\ \text{“(”}\ (\textit{VarList})?\ \text{“)”} \\
 & & \textbf{begin}\ \textit{Stmt}\ \text{“;”}\ \textbf{end} \\
\textit{FuncCall} & ::= & \textit{Func}\ \text{“(”}\ \textit{ExprList}?\ \text{“)”} \\
\textit{Program} & ::= & (\textit{VarList}\ \text{“;”})?\ \textit{FuncDef}^+
\end{array}
$$

Figure 3.1: The core syntax of our language

```
1: x, y, z;
2: main() begin
3:  r1 = load &x;
4:  r2 = r1 * 0;
5:  if r2 == 0 then
6:   store &y, 1;
7:   r3 = 0
8:  else r4 = 1 fi
9:  r5 = φ(r3, r4);
10: store &z, r5;
11:end
```

Figure 3.2: An example code snippet written in our language

(a) Data dependency	(b) Explicit control dependency	(c) Address dependency	(d) Implicit control dependency

`r1 = load &x;`
`r2 = r1 * 0;`
`store &y, r2;`
	`r1 = load &x;`
	`if r1 != 0 then`
	`  store &y, 1`
	`else`
	`  skip`
	`fi;`
		`r1 = load &x;`
		`store r1, 0;`
		`r2 = load &z;`
		`store &y, r2;`
			`r1 = load &x;`
			`if r1 != 0 then`
			`  store &y, 1`
			`else`
			`  store &z, 0`
			`fi;`
			`r2 = load &z;`
			`store &y, r2;`

Figure 3.3: Does the last store to `y` depend on the first load "`r1 = load &x`" in each of theses examples? Assume `z=1` before each execution.

definition of the notion of dependency for our language with the operational semantics is shown below:

$$\text{Const.Expr:}\frac{}{\langle const, V, dep, D, PC, FD\rangle \rightarrow \langle\langle const, \emptyset\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Var.Expr:}\frac{}{\langle var, V, dep, D, PC, FD\rangle \rightarrow \langle\langle V[var], D[var]\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Unary.Expr:}\frac{\langle E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}{\langle op_u\ E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle op_u\ \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Binary.Expr:}\frac{\begin{array}{c}\langle E_1, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_1, \mathcal{D}_1\rangle, V', dep', D', PC', FD'\rangle \\ \langle E_2, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle \mathcal{V}_2, \mathcal{D}_2\rangle, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\langle E_1\ op_b\ E_2, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_1\ op_b\ \mathcal{V}_2, \mathcal{D}_1 \cup \mathcal{D}_2\rangle, V'', dep'', D'', PC'', FD''\rangle}$$

$$\text{Load.Expr:}\frac{\langle Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Addr}}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle \quad \mathcal{V}_{\text{load}} = \text{load}(\mathcal{V}_{\text{Addr}})}{\langle \text{load } Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{load}}, \mathcal{D} \cup \{fresh\_load\}\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Assignment:}\frac{\langle E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}{\langle var = E, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V'[var := \mathcal{V}], dep', D'[var := \mathcal{D}], PC', FD'\rangle}$$

$$\text{Store:}\frac{\begin{array}{c}\langle Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Addr}}, \mathcal{D}_{\text{Addr}}\rangle, V', dep', D', PC', FD'\rangle \\ \langle Val, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Val}}, \mathcal{D}_{\text{Val}}\rangle, V'', dep'', D'', PC'', FD''\rangle \quad s := \text{store}(\mathcal{V}_{\text{Addr}}, \mathcal{V}_{\text{Val}})\end{array}}{\begin{array}{l}\langle \text{store } Addr, Val, V, dep, D, PC, FD\rangle \rightarrow \\ \quad \langle skip, V'', dep'' \cup ((\mathcal{D}_{\text{Addr}} \cup \mathcal{D}_{\text{Val}} \cup PC'' \cup FD'') \times \{s\}), D'', PC'', FD'' \cup \mathcal{D}_{\text{Addr}}\rangle\end{array}}$$

$$\text{Composition.Skip:}\frac{}{\langle skip; S, V, dep, D, PC, FD\rangle \rightarrow \langle S, V', dep', D', PC', FD'\rangle}$$

$$\text{Composition.Left}\frac{\langle S_1, V, dep, D, PC, FD\rangle \to \langle S_1', V', dep', D', PC', FD'\rangle}{\langle S_1; S_2, V, dep, D, PC, FD\rangle \to \langle S_1'; S_2, V', dep', D', PC', FD'\rangle}$$

$$\text{Phi.Taint:}\frac{}{\langle\langle v, \mathcal{D}\rangle, V, dep, D, PC, FD\rangle \to \langle skip, V, dep, D[v := (D[v] \cup \mathcal{D})], PC, FD\rangle}$$

$$\text{assign\_phi}(v = \phi(v_0, v_1); phiList, \mathcal{V}_P) \Rightarrow v = v_{\mathcal{V}_P}; \text{assign\_phi}(phiList, \mathcal{V}_p)$$

$$\text{assign\_phi}(\epsilon, \mathcal{V}_P) \Rightarrow \epsilon \qquad \text{assign\_phi}(;, \mathcal{V}_P) \Rightarrow \epsilon$$

$$\text{taint\_phi}(v = \phi(v_0, v_1); phiList, \mathcal{D}) \Rightarrow \langle v, \mathcal{D}\rangle; \text{taint\_phi}(phiList, \mathcal{D})$$

$$\text{taint\_phi}(\epsilon, \mathcal{V}_P) \Rightarrow \epsilon \qquad \text{taint\_phi}(;, \mathcal{V}_P) \Rightarrow \epsilon$$

$$\text{If.True:}\frac{\begin{array}{c}\langle cond, V, dep, D, PC, FD\rangle \to \langle\langle true, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle \\ \langle S_1, V', dep', D', PC' \cup \mathcal{D}, FD'\rangle \to \langle skip, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\begin{array}{l}\langle \text{if } cond \text{ then } S_1 \text{ else } S_2 \text{ fi } phi, V, dep, D, PC, FD\rangle \to \\ \quad \langle \text{assign\_phi}(phi; , 0) \text{ taint\_phi}(phi; , \mathcal{D})\ skip, V'', dep'', D'', PC, \\ \quad FD' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S_2)\}\rangle\end{array}}$$

$$\text{If.False:}\frac{\begin{array}{c}\langle cond, V, dep, D, PC, FD\rangle \to \langle\langle false, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle \\ \langle S_2, V', dep', D', PC' \cup \mathcal{D}, FD'\rangle \to \langle skip, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\begin{array}{l}\langle \text{if } cond \text{ then } S_1 \text{ else } S_2 \text{ fi } phi, V, dep, D, PC, FD\rangle \to \\ \quad \langle \text{assign\_phi}(phi; , 1) \text{ taint\_phi}(phi; , \mathcal{D})\ skip, V'', dep'', D'', PC, \\ \quad FD'' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S_1)\}\rangle\end{array}}$$

$$\text{While.Taken:}\frac{\begin{array}{c}\langle \text{assign\_phi}(phi; , 0)\ skip, V, dep, D, PC, FD\rangle \to \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \to \langle\langle true, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle S, V'', dep'', D'', PC'' \cup \mathcal{D}, FD''\rangle \to \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle \text{while } phi\ cond \text{ do } S \text{ od}, V, dep, D, PC, FD\rangle \to \langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC\rangle, \\ \quad V''', dep''', D''', PC''', FD'''\rangle\end{array}}$$

$$\text{While.Untaken:}\frac{\begin{array}{c}\langle \text{assign\_phi}(phi; , 0)\ skip, V, dep, D, PC, FD\rangle; \to \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \to \langle\langle false, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle \text{taint\_phi}(phi; , \mathcal{D})\ skip, V'', dep'', D'', PC'', FD''\rangle \to \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle \text{while } phi\ cond \text{ do } S \text{ od}, V, dep, D, PC, FD\rangle \to \\ \quad \langle skip, V''', dep''', D''', PC, FD''' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S)\}\rangle\end{array}}$$

$$\text{Loop.Taken:}\frac{\begin{array}{c}\langle \text{assign\_phi}(phi; , 1)\ skip, V, dep, D, PC, FD\rangle \to \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \to \langle\langle true, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle S, V'', dep'', D'', PC'' \cup \mathcal{D}, FD''\rangle \to \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V, dep, D, PC, FD\rangle \to \\ \quad \langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V''', dep''', D''', PC''', FD'''\rangle\end{array}}$$

$$\text{Loop.Untaken:}\frac{\begin{array}{c}\langle \text{assign\_phi}(phi;,1)\ skip, V, dep, D, PC, FD\rangle \to \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \to \langle\langle false, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle \text{taint\_phi}(phi;,\mathcal{D})\ skip, V'', dep'', D'', PC'', FD''\rangle \to \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V, dep, D, PC, FD\rangle \to \\ \quad \langle skip, V''', dep''', D''', PC_{old}, FD''' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S)\}\rangle\end{array}}$$

We formalize the program execution state as the tuple $\delta = \langle N, V, dep, D, PC, FD\rangle$, where N represents a computational node (e.g., an expression or statement), V represents a mapping from an expression to its concrete value, *dep* represents a dependency set, which is a subset of the Cartesian product of the load set and store set in an execution. For example, "$(l, s) \in$ *dep*" means that store s depends on load l; D represents a dependency mapping from an expression to the set of loads the expression depends on, PC represents the set of loads on which the current instruction has explicit control dependency, and FD represents the set of loads on which future stores should depend. Essentially, the rules of our semantics focus on recording which loads an expression or statement depends on in each step of the execution, and when we finish executing the program, the final result is recorded in *dep* — the complete dependency relation between loads and stores in the execution. The details of dependency rules follow:

1. Expressions: In general, an expression has data dependency on its subexpressions, meaning that the expression depends on whatever loads its subexpressions depend on. In our operational semantics, an expression E can be reduced to a pair $\langle \mathcal{V}, \mathcal{D}\rangle$, in which $\mathcal{V}$ represents the concrete value to which E is evaluated, and $\mathcal{D}$ represents the set of loads E depends on. The *Const.Expr* rule means a constant numeral is evaluated to itself and does not depend on any loads; the *Var.Expr* rule means a variable *var* retrieves its concrete value recorded in V and its dependency set recorded in D; the *Unary.Expr* and *Binary.Expr* rules are specifically for unary and binary expressions, respectively; the *Load.Expr* rule means that a load expression has data dependency on its address. It is important to note that in the dependency relation we define, loads

are the sources and stores are the sinks. More specifically, given a load instruction l and its address $Addr$ (which depends on $\mathcal{D}$), l depends on the union of $\mathcal{D}$ and l itself, denoted as $fresh\_load$ (since l becomes the source in the dependency relation after we execute the load). In the *Load.Expr* rule, "$\mathcal{V}_{load} = \text{load}(\mathcal{V}_{Addr})$" means reading the value from address $\mathcal{V}_{Addr}$ and assigning it to $\mathcal{V}_{load}$.

2. Assignments: For an assignment statement "$var = E$", the left-hand side variable var has data dependency on the right-hand side expression E. Thus, the *Assignment* rule assigns the concrete value of E to var and also passes the dependencies of E to var.

3. Stores: For a store statement s, "`store Addr, Val`", where the address `Addr` has dependency on load set $\mathcal{D}_{Addr}$ and the storing value has dependency on load set $\mathcal{D}_{Val}$, s has data dependency on $\mathcal{D}_{Addr} \cup \mathcal{D}_{Val}$. For example, in 3.3 (a), since the storing value `r2` has data dependency on "`r1 = load &x`", so store "`store &y, r2`" depends on "`r1 = load &x`". In addition, s should depend on the set of loads on which it has explicit control dependency, which in our operational semantics is recorded in PC. For example, in 3.3 (b), store "`store &y, 1`" depends on "`r1 = load &x`" because of explicit control dependency.

   However, it is not sufficient to only consider explicit data and control dependencies. Consider the question of whether the last store "`store &y, r2`" depends on the first load "`r1 = load &x`" in Figure 3.3 (c). At first glance, it may appear to be that "`store &y, r2`" only depends on "`r2 = load &z`" but is independent of "`r1 = load &x`" since "`store &y, r2`" does not have an explicit data dependency or control dependency on "`r1 = load &x`". However, if we consider the question of what value the memory location `y` will hold at the end of the execution (assuming `z=1` before each execution), we can see that the answer becomes either the value `1` or `0` depending on whether `r1` points to the memory address of `z`, which means "`store &y, r2`" actually depends on "`r1 = load &x`". Technically speaking, if we

can tell by some static analysis (e.g., some sort of points-to analysis) that `r1` always points to a different memory address than `z`, then "`store &y, r2`" does not depend on "`r1 = load &x`". However, if we rely on such analysis in our rules, we will end up with an extremely complicated dependency notion; moreover, the fact that C/C++ allows pointer arithmetics and pointer conversions would further complicate the dependency notion because the precise addresses of loads and stores may not be always known. Hence, instead of incorporating the details of the points-to analysis in our definition of dependency, we take a very conservative approach. In our semantics, if the address of a store depends on some load, then we require that all subsequent stores also depend on that load. We refer to this type of dependency as an *address dependency*. Back to the example shown in Figure 3.3 (c), we simply conservatively say that the store "`store &y, r2`" depends on the load "`r1 = load &x`", even if the compiler knows `r1` will never point to the address of `z`. As a result, in our implementation, if the compiler wishes to reorder a store s' ("`store &y, r2`" in this example) up above another store s ("`store r1, 0`" in this example), it has to make s' depend on any loads that the address of s depends on ("`r1 = load &x`" in this example).

Consider the example shown in Figure 3.3 (d), in which we need to answer the same question of whether the last store "`store &y, r2`" depends on the first load "`r1= load&x`". Similar to the example shown in Figure 3.3 (c), the store "`store &y, r2`" does not have a data dependency or explicit control dependency on "`r1 = load &x`"; however, given `z=1` initially, the value that the memory location `y` will hold can actually be value `1` or `0`, depending on whether the condition "`r1 != 0`" is true. The essential reason why we have such dependency is that in an untaken branch there exists a store (i.e., "`store &z, 0`" in the `else` branch) which overwrites a memory location that is later read. Similar to address dependency, a fine-grained definition of this type of dependency would require the introduction of a program analysis and complicate our dependence notion. Instead, we take a conservative approach by stating that if the

condition of a control flow block (i.e., an `if/else` block or `while` loop) depends on a load, and the untaken branch has a syntactically reachable store, then all subsequent stores after the conditional branch also depend on that load. We refer to this type of dependency as *implicit control dependency*.

We can see that the address dependency set and implicit control dependency set share two commonalities: (1) along with the execution of a program, both dependency sets will only be augmented by adding more loads; and (2) whenever a future store statement is executed, we must ensure that the future store depends on the loads in these two sets. Hence, our operational semantics uses FD to record the union of address dependency and implicit control dependency, on which all future stores depend.

Figure 3.4 presents a dependency cycle example involving address dependencies. In this example, global variables `x` and `y` are `0` and each element in the global array `z` is `0`. For the problematic execution in which `r1=r2=1`, the reason why `r1=x` can return value `1` is a chain of justifications that start with the assumption that `r1=x` can return value `1`. As a result, the load in line 3 of Thread 1 does not read from the store in line 2 of Thread 1. Note that the dependence is transmitted through the absence of reading from the store in line 2. Rather than explicitly model the dependence from the store to the load, our semantics leverages the fact that all dependency chains end with a store and simply adds a dependency on the load "`r1=x`" to all stores after the store "`z[r1]=1`", which in our example is the store "`y=1`". Our dependency-preserving memory model forbids this execution because there exists a cycle in $dep \cup rf$.

4. Phi (ϕ) functions: A phi function "`v=`ϕ`(`v_0`,`v_1`)`" with respect to condition *cond* is essentially an assignment statement that selects its right-hand side value associated with *cond*, whether *cond* is from an `if/else` branch or a `while` loop. For an `if/else` branch, v_0 comes from the `if` branch, and v_1 comes from the `else` branch. For a `while` loop, v_0 comes from outside the loop, and v_1 comes from inside the loop. For example, if

int x=y=0; // Initially 0
int z[2]; // Initially 0	

Thread 1	Thread 2
1:r1 = x; 2:z[r1] = 1; 3:if (!z[0]) 4: y = 1;	1:r2 = y; 2:x = r2;

Figure 3.4: If x=y=0 and each element in array z is 0 initially, can r1=r2=1? Note that according to our dependency notion, store "y=1" has an address dependency on load "r1=x" because the address of store "z[r1]=1" depend on "r1=x".

the phi function is associated with an `if/else` branch, and condition *cond* is true, then the phi function effectively becomes "$v=v_0$" with an extra (explicit) control dependency on *cond*. Thus, the phi variable v depends on whatever loads v_0 and *cond* depend on. In our rules, the *assign_phi* function transform phi functions to assignments so that we can apply the *Assignment* rule for data dependency, and the *taint_phi* function and the *Phi.Taint* rule together taint the phi variables with the explicit control dependency on the condition. For example, the phi variable r5 in line 9 in Figure 3.2 has an explicit control dependency on the `if` condition, which depends on the load "`r1 = load &x`". As a result, by the *Store* rule, the store "`store &z, r5`" in line 10 also depends the "`r1 = load &x`".

5. If/else branches: An `if/else` branch can potentially introduce explicit and implicit control dependencies, as shown in Figure 3.3 (b) and (d). In addition, we must ensure that the phi variables associated with the branch also have dependencies on the appropriate right-hand side variable and a dependency on the `if` condition as discussed above. The *If.True* and *If.False* rules are applied when the `if` condition is true or false, respectively. The `hasReachableStore()` function returns true or false for whether or not a given block of statements has a syntactically reachable store (i.e., potential store). Note that the explicit control dependency set PC returns to its original state after we execute the `if/else` branch, and we track implicit control dependencies by applying

the `hasReachableStore()` function on the untaken branch.

6. While loops: In our dependency rules, a `while` loop can be unrolled indefinitely and viewed as if they were nested `if/else` branches. However, in terms of formalization, unlike normal `if/else` branches, we need to distinguish the first time we enter the `while` loop from later loop continuations for two reasons: (1) we need to assign the phi functions differently depending on whether we enter the loop for the first time or not; and (2) when we finish executing a loop, since we need to recover the explicit control dependency set PC, we need to record the old PC set based on the two different cases. Thus, we define the *While.Taken* rule for cases where we enter the loop for the first time and the loop condition is `true`, and the *While.Untaken* rule for those where we enter the loop for the first time and the loop condition is `false`. Note that once a `while` loop is taken for the first time, we change the loop keyword from `while` to `loop` as an indicator that the loop has been taken at least once. Also, when a `while` loop is taken for the first time, we record the old PC so that we can recover the PC status when we finish executing the loop. We then define the *Loop.Taken* rule for cases where a loop is taken after the first time, and the *Loop.Untaken* rule for cases where a loop is finished after being executed at least once. Note that applying any of these four rules has an effect on the explicit control dependency set PC and future store dependency FD similar to that of the conditional branch rules.

7. Function calls: Since we only have function calls in which the functions are pre-defined and have a definite function name, a function call can be viewed as an inlined block of statements with extra data dependencies from the actual parameters to the formal parameters and from the return value to the actual function call result. Hence, to simplify the presentation and focus on the core problem, we omit the dependency rule involving function calls in the operational semantics shown above.

   However, this is not sufficient for real-world programming languages, which can have

function pointers (e.g., C/C++) or virtual dispatch mechanisms (e.g., object-oriented programming languages). The essential problem is that when the address of a function call depends on some load l, we should conservatively assume that the function call could potentially have stores that write to any possible memory location and thus must ensure that any future store from the point of the function call also depends on load l. We refer to this type of dependency as a *function dependency*. It is important to note that the implementation of our dependency-preserving compiler shown in Chapter **??** effectively enforces function dependency.

3.2.1 Out-of-Thin-Air Properties of the Dependency-Based Memory Model

Since there is no agreed upon definition of out-of-thin-air executions, we provide a proof sketch for a property about the causality of executions in our memory model.

Definition 3.2.1. *(Value independent semantics). A memory model has value-independent semantics iff the semantics of the memory model do not depend on the value loaded or stored with the exception of CAS. The C/C++ and Java memory models are both value independent.*

Definition 3.2.2. *(Load available semantics). A memory model has load available semantics if a load can always read from some value that will not affect which values later loads can read from. For the C/C++ memory model, this is the earliest store in the modification order that is visible to the load.*

Theorem 3.2.1 (Dependency Theorem)**.** *For a memory model that has value-independent and load available semantics and that ensures that $dep \cup rf$ is acyclic, then if a store s is not reachable from a load l in the graph $dep \cup rf$ for an execution e, then for any value v that the load l returns there exist an execution e' with an equivalent load which returns value v*

such that either: (1) e' has an error or (2) e' has a store s' that writes the same value to the same address as s.

Proof Sketch.

Define A to be the part of the execution that can reach l in the $dep \cup rf$ graph. Define B to be the part of the execution that l can reach in the $dep \cup rf$ graph. Define C to the part of the execution that can reach s in the $dep \cup rf$ graph.

Then:

1. Load dependencies for the address of a store s_a or the condition of a branch with an untaken store in B is not sb before anything in C. This is true by the definition of dependency.

2. Load dependencies for the address of a store s_a or the condition of a branch with an untaken store in B is not sb before anything in A. This is true by the definition of dependency and by the assumption that $dep \cup rf$ is acyclic.

3. There is no load in C that reads from any store in B. This is true by the definition of dependency.

4. Load dependencies for the address of a store s_a or the condition of a branch with an untaken store that are sb before A are in A. This is true by the definition of dependency.

5. Load dependencies for the address of a store s_a or the condition of a branch with an untaken store that are sb before C are in C. This is true by the definition of dependency.

For any value v that load l returns, we can construct an execution e' in which (1) every store that is sb before $A \cup C$ in the execution e' has an equivalent store in e that writes to the

same address, (2) every store in $A \cup C$ in e is in e' and writes to the same address, and (3) every load that is sb before $A \cup C$ reads from the same store as it did in e (note that some loads than are sb, but not in $A \cup C$ may be missing) since all of the load dependencies for the conditional branches or addresses of stores are in $A \cup C$. The stores in execution e' that are sb before $A \cup C$ are a subset of stores in execution e and they write to the same addresses so this is possible (loads in execution e' who are missing their corresponding store are not in $A \cup C$ and can simply be made to read from some store without affecting other loads since we assume that the memory model has load available semantics). Note that the stores may not write the same values, but the memory semantics are value-independent and thus admit the same rf relation. Note that we may have new loads appear that are sb before $A \cup C$, but such loads can always read from a value by the assumption that the memory model has load available semantics.

The execution e' may throw an error in which case we trivially prove the property. Thus assume that execution e' does not throw an error. Then by induction on $dep \cup rf$ and the definition of dep, the store s' must store the same value as store s.

Theorem 3.2.1 implies that executions with causality cycles or satisfaction cycles in which a store s cyclically justifies the value it stores are not possible if $dep \cup rf$ is acyclic. Any load l that reads from s cannot reach s in the $dep \cup rf$ graph since it is acyclic and the load l reads from s. Thus by the theorem, the load l can return any value and store s will still store the same value.

3.3 A Load-Store-Order-Preserving Memory Model

An alternative approach to eliminating out-of-thin-air behaviors is to strengthen the existing C/C++ memory model by requiring *sequence-before* $\cup$ *reads-from* to be acyclic. This

well-known C/C++ memory model variant has been proposed by researchers (Vafeiadis and Narayan, 2013; Batty et al., 2013; Boehm and Demsky, 2014) as one of the possible approaches to forbidding out-of-thin-air behaviors. Note that this is not the "perfect" fix to the problem since it forbids not just the problematic out-of-thin-air executions (e.g., Figure 2.2) but also some legitimate executions such as the load buffering example shown in Figure 2.1. Notably, this approach is less likely to be acceptable for Java-like memory models because they must describe the behavior of all loads and stores (including `volatile` and non-volatile memory accesses), which is likely to incur a much higher cost[2].

[2]To implement the load-store-order-preserving memory model discussed here, a Java compiler (JVM) must ensure that no compiler optimizations will effectively perform the load-store reordering for all loads and stores and also insert load-store fences/dependencies in the generated code accordingly.

Chapter 4

Dependency-Preserving Compiler

This chapter describes the design and implementation of our approach to preserving our extended memory model in the LLVM compiler infrastructure (Lattner and Adve, 2004). We target the LLVM compiler in this dissertation for two reasons: (1) LLVM is widely supported and considered by many as the state-of-the-art compiler framework; and (2) LLVM is not just adopted as a C/C++ compiler but is also adopted in the context of a commercial JVM, e.g., Azul's Falcon compiler (Azul, 2017).

4.1 Design

The LLVM compiler infrastructure is designed to compile source code and generate optimized library or executable files in a modular and reusable fashion. The standard LLVM compilation pipeline is shown in Figure 4.1, and we illustrate the workflow as follows:

1. Given C/C++ source code files, the *Clang* front end translates them into a type of target-independent intermediate representation (IR), i.e., the LLVM Bitcode or the

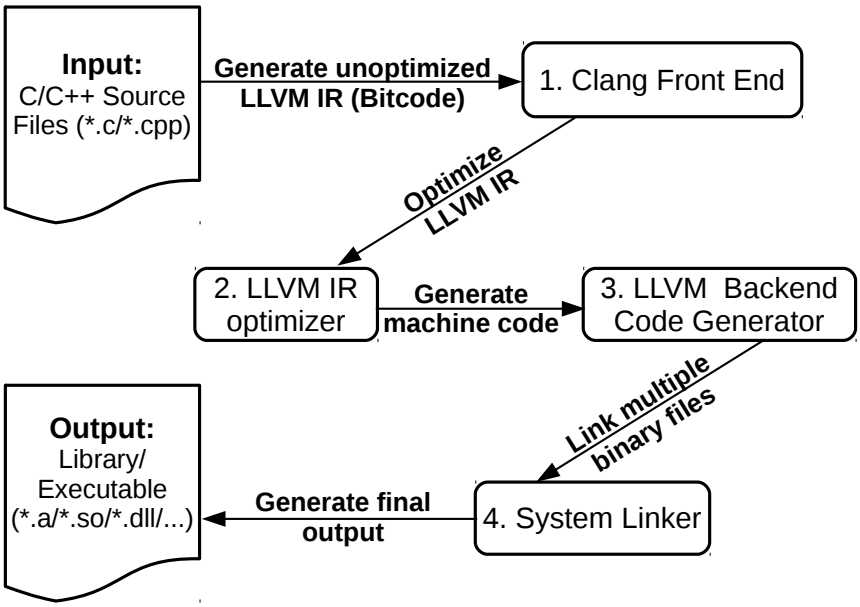

Figure 4.1: The standard LLVM compilation workflow for C/C++

LLVM IR. The LLVM IR generated in this step has not been optimized yet, and hence it preserves all the trivial computations and control flows except very obvious constant folding, etc. For example, the statement "x = 2 * 2" in the source code will be translated into "store 4, x" in the unoptimized IR; however, the statement "x = r1 * 0" will be preserved. It is important to note that such trivial constant folding does not break dependencies.

2. The unoptimized LLVM IR generated in step 1 will then go through the LLVM IR optimizer, which performs a list of target-independent LLVM IR transformation passes to generate optimized LLVM IR. In the LLVM tool chain, the LLVM optimizer — *opt* — can perform these optimizations. Note that these transformation passes can potentially change the IR and break dependencies.

3. The optimized LLVM IR generated in step 2 will then be passed to the LLVM backend code generator, which performs instruction selection, register allocation, peephole optimizations, etc., and finally generates optimized binary object files. In the LLVM tool

chain, the LLVM static compiler — *llc* — implements this step. Note that the backend transformations are implemented as a list of code generation passes (LLVM machine passes performed in some machine code representations), and these passes can also potentially break dependencies. In general, the LLVM infrastructure supports multiple backend code generators for different architectures. In our work, we use the AArch64 (ARM's 64-bit architecture) backend as a case study and thus will focus on the code generation passes for AArch64 hardware. Our case study explore ARM 64 because it is the most relevant mainstream processor that implements a memory model that does not preserve load-store ordering. Intel processors implement the TSO memory model which by default preserves load-store ordering, and thus are expected to incur significantly less overhead to implement our memory model.

4. Finally, the system linker will link the optimized object files and generate optimized library or executable files. Note that on a Linux machine, it is a common case to use the GNU linker as the default system linker.

As shown in the above pipeline, in order to ensure that dependencies are preserved down to the generated binary code, we must make sure the transformations in step 2 and 3 preserve the dependencies. Ideally, we should review each transformation pass. If it can potentially break dependencies, we should modify the optimization to retain most of the benefits of the optimization while preserving dependencies; otherwise, we should leave the pass unchanged. However, for our research project, this incurs too much manual effort. There are more than 50 LLVM transformation passes in step 2. Fortunately, a preliminary result shows that if we run only a select set of 35 IR-to-IR passes alone in step 2, the performance loss over running the standard set of IR-to-IR passes (as called in "opt -O3") is only 1.8%. As a result, we disable all other IR-to-IR passes and focus our efforts on the select set of passes. We show the set of IR passes that we enable in Appendix A.

4.2 Implementation

Several important compiler optimization passes are inherently dependency-preserving and require either no changes or only minor changes. We first discuss those that require no changes and then discuss how we modify/disable the remaining passes to preserve dependencies.

4.2.1 Unmodified Passes

Function inlining: This optimization expands specific function call sites in the body of the caller functions, potentially reducing function call overhead and introducing more opportunities for later optimizations. Our preliminary result shows that turning off inlining incurs an overhead of 26% on AArch64 targets. Fortunately, no change is required for this pass as long as we conservatively preserve the syntactic dependencies in all functions. For example, Figure 4.2 shows a function `foo` that internally calls another function `bar`. The third column shows that as long as we preserve the dependency between the argument `arg` and the return statement "`return arg * 0`", the `foo` function after inlining still preserves the dependency between the load from `x` and the store to `y`.

Original `foo()`	Original `bar()`	Inlined `foo()`

`void foo() {`
`  r1 = x;`
`  y = bar(r1);`
`}`
	`int bar(int arg) {`
	`  return arg * 0;`
	`}`
		`void foo() {`
		`  r1 = x;`
		`  y = r1 * 0;`
		`}`

Figure 4.2: Function inlining does not break the dependency between the load from `x` and the store to `y` as long as function `bar` preserves its internal dependencies, as shown in the third column.

Common subexpression elimination: Common subexpression elimination (CSE) replaces a redundant expression with the value of a pre-computed common expression. For

example, it will transform the instructions "`z=x*y; t=x*y`" to "`r1=x*y; z=r1; t=r1`". We can see that the dependency from `x` and `y` to `t` is preserved because it is carried by the intermediate value `r1`. In LLVM, the global value numbering (*gvn*) pass can perform redundant load elimination that has similar effect to CSE.

Dead code elimination: Dead code elimination in general eliminates the instructions that are unreachable or have no visible effects to the program, and does not break dependencies. In LLVM, this corresponds to *adce* (aggressive dead code elimination) and *dce* (dead code elimination).

4.2.2 Modified Passes

To preserve a simple notion of syntactic dependency, we must consider both data dependencies and control dependencies. For example, as shown in Figure 4.3, the original LLVM optimizations (`-O3`) recognize that the expression "`r1*0`" will always generate the value `0` and will transform the store instruction to "`y=0`", which no longer depends on the load from `x`. In the other example shown in Figure 4.4, the original LLVM optimizations (`-O3`) determine that no matter which branch the program takes, it will execute the same store instruction, so it merges the two stores to `y` and later eliminates the empty control blocks. Hence, this breaks the dependency from "`y=1`" to the load from `x`.

Unoptimized code	Optimized code
`r1 = x;`	`r1 = x;`
`y = r1 * 0;`	`y = 0;`

Figure 4.3: LLVM optimizations (`-O3`) can break data dependencies.

We next outline the important optimization passes that we have modified to preserve dependencies:

Combining redundant instructions (*instcombine*): This pass combines instructions

Unoptimized code	Optimized code
`if (x > 0) y = 1;` `else y = 1;`	`y = 1;`

Figure 4.4: LLVM optimizations (`-O3`) can break control dependencies.

to fewer and simpler ones and does not modify the control flow graph. For example, it performs simple constant folding, dead code elimination, algebraic simplification, and re-ordering of operands to expose more common subexpression elimination opportunities, etc. To preserve dependencies, we modify this pass to disable the transformations that can potentially break dependencies. Figure 4.5 shows examples of our modification to the instruction simplification optimization in order to preserve dependency. More specifically, Figure 4.5(a) shows an example in which we prevent it from simplifying the condition "`r2=(r1==r1)`" to "`r2=true`". At the same time, we still allow those dependency-preserving transformations and also perform a limited form of strength reduction on algebraic instructions when the original simplification would break dependencies. For example, Figure 4.5(b) shows that although we cannot completely simplify the three `AND` instructions to the value `0`, we can still perform partial simplification and eliminate two redundant `AND` instructions; Figure 4.5(c) shows that while we cannot transform "`r1*0`" to the value `0`, we can transform it to a potentially less expensive `AND` instruction.

	Unoptimized code		Dependency-preserving code
(a)	`r2 = (r1 == r1);` `if (r2)...`	⇒	`r2 = (r1 == r1);` `if (r2)...`

(b)	`r2 = r1 & 0xffff;`
	`r3 = r1 & 0xffff0000;`
	`r4 = r2 & r3;`
		⇒	`r4 = r1 & 0;`

(c)	`r2 = r1 * 0;`	⇒	`r2 = r1 & 0;`

Figure 4.5: Examples of how the dependency-preserving *instcombine* pass transforms the code.

Simplify the CFG (*simplifycfg*): This pass simplifies control flows, which includes a form of dead code elimination with respect to control flows (e.g., removing unreachable basic

blocks and basic blocks that contain only an unconditional branch), basic block merging, and hoisting common code outside of control flow blocks. Since we disable dependency-breaking algebraic simplifications in the *instcombine* pass, the *simplifycfg* pass cannot eliminate control flows by statically calculating the value of conditions. For example, in Figure 4.6 (a), since we disable simplifying the condition "`r1 == r1`" to the value `true`, the transformation shown in the middle column cannot happen, and thus the control flow dependency is preserved.

However, transformations that involve moving stores out of the control flow blocks are generally problematic and should be prohibited. Figure 4.4 shows such an example in which the pass first hoists the common stores "`y=1`" out of the `if/else` branch and then eliminates the `if/else` blocks, which breaks the dependency of "`y=1`" on the load from `x`. Figure 4.6 (b) shows an even more problematic example. Before the transformation, the store "`y=1`" depends on the load "`r1=x`", and the store "`*addr=2`" also depends on "`r1=x`" because of the conditional store "`y=1`" that is sequenced-before it (i.e., implicit control dependency). After the original transformation, although the new unconditional store "`y=r1?1:0`" still depends on "`r1=x`", the later store "`*addr=2`" no longer depends on "`r1=x`". Our dependency-preserving transformation preserves this missing dependency by adding redundant computations that require the value of the condition (i.e., "`(&y)|(r1&0)`") to compute the address of the new unconditional store (i.e., "`*r2=r1?1:0`") so that all later stores still depend on the old condition `r1`.

We also disable the elimination of control flow blocks in some cases even when there is no store within the control flow blocks. For example, in Figure 4.6 (c), we can see in the unoptimized code that the store to `z` syntactically depends on the load from `x` even though `r1` is a local variable. However, before LLVM runs the *simplifycfg* pass, it runs a pass that transforms the LLVM IR to static single assignment (SSA) form, which simplifies the instruction "`z=r1`" to "`z=1`" and makes the `if/else` blocks empty. This step alone preserves

the control dependency because it does not modify the control flow. However, after that, the *simplifycfg* pass will eliminate the empty `if/else` blocks and transform it to the code shown in the middle, in which "`z=1`" does not depend on the load from `x` anymore. Our dependency-preserving transformation on the right keeps the empty conditional branch.

This shows that multiple dependency-preserving passes combined together can break dependencies. The fundamental problem that causes this is the poorly defined notion of dependency in the IR. In the example in Figure 4.6 (c), the later *simplifycfg* pass has no information about whether the empty conditional branch carries a dependency to later stores, thus eliminating it can potentially break a dependency, as shown in this case. Ideally, if the IR was augmented with extra dependency edges between statements, we could use that information to ensure that a specific transformation does not break existing dependency edges. In practice, to make our approach simpler to implement in the existing LLVM framework without requiring large changes to LLVM's IR, we adopt a coarse-grained extension to the IR such that basic blocks contain extra information indicating whether some other statements may or may not depend on them. In this case, when we construct the SSA form and encounter a PHI node that has the same value from multiple basic blocks, we conservatively mark the incoming blocks as unremovable to preserve such control flow blocks even if they are empty.

Passes that potentially reorder stores: According to our dependency notion, reordering an earlier store s_1 and a later store s_2 can potentially break address dependencies if the address of s_1 depends on some load that s_2 does not depend on. We list the passes that can reorder stores and our corresponding strategies as follows:

1. Dead store elimination pass (*dse*): This pass looks for stores that have no visible side effects and eliminates them. Figure 4.7 (a) is an example in which the first store in the unoptimized code "`arr[r1]=0`" is a dead store since there are no loads after it until the last store "`arr[r1]=1`". In addition, store "`y=1`" depends on load "`r1=x`" because

	Unoptimized	Original transformation	Dependency-preserving transformation

(a)	`r1 = x;`
	`if (r1 == r1) y = 1;`
	`else y = 2;`
		`r1 = x;`
		`y = 1;`
			`r1 = x;`
			`if (r1 == r1) y = 1;`
			`else y = 2;`

(b)	`r1 = x;`
	`y = 0;`
	`if (r1) y = 1;`
	`*addr = 2;`
		`r1 = x;`
		`y = r1 ? 1 : 0;`
		`*addr = 2;`
			`r1 = x;`
			`r2 = (&y) | (r1 & 0);`
			`*r2 = r1 ? 1 : 0;`
			`*addr= 2;`

(c)	`r1 = 0;`
	`if (x > 0) r1 = 1;`
	`else r1 = 1;`
	`z = r1;`
		`z = 1;`	`// Keep empty blocks`
			`if (x > 0) ;`
			`else ;`
			`z = 1;`

Figure 4.6: Examples of how the *simplifycfg* pass can potentially break dependencies.

the address of "`arr[r1]=0`" depends on "`r1=x`" (i.e., address dependency). After the original transformation, this pass eliminates the store "`arr[r1]=0`", which breaks the dependency from store "`y=1`" to load "`r1=x`". Our solution is to add redundant computations involving the address `arr[r1]` to the store to `y`, which ensures that the dependency on "`r1=x`" is passed to the store "`*r2=1`".

2. Loop invariant code motion (*licm*): This pass optimizes loops by moving loop invariant code outside of the loop. In general, it hoists load instructions out of the loop body and sinks store instructions to the end of the loop. Figure 4.7 (b) shows an example that illustrates why this can be problematic. In the unoptimized code, all stores "`arr[i++]=1`" depend on the load from `x`, but the original transformation breaks this dependency by sinking the store "`addr[x]`=0". We disable such transformations.

3. SLP (superword-level parallelization) vectorization (*slp-vectorizer*): This pass can combine similar independent instructions into vector instructions. Figure 4.7 (c) shows how it can effectively reorder stores by combining adjacent stores. We modify this pass to prohibit the original transformation shown in Figure 4.7 (c).

4. Memory copy optimization (*memcpyopt*): This pass performs optimization related to

memset, memcpy, and memmove calls, and we disable this pass.

	Unoptimized	Original transformation	Dependency-preserving transformation

(a)	`r1 = x;`
	`arr[r1] = 0;`
	`y = 1;`
	`arr[r1] = 1;`
		`r1 = x;`
		`y = 1;`
		`arr[r1] = 1;`
			`r1 = x;`
			`r2 =(&y)|((arr+r1)&0);`
			`*r2 = 1; // y = 1`
			`arr[r1] = 1;`

(b)	`do {`
	`addr[x] = 0;`
	`arr[i++] = 1;`
	`} while (i < 100);`
		`do {`
		`arr[i++] = 1;`
		`} while (i < 100);`
		`addr[x] = 0;`
			`do {`
			`addr[x] = 0;`
			`arr[i++] = 1;`
			`} while (i < 100);`

(c)	`arr[x&0] = 0;`
	`y = 1;`
	`arr[1] = 1;`
	`arr[2] = 2;`
	`arr[3] = 3;`
		`y = 1;`
		`arr[0..3] = {0..3};`
			`arr[x&0] = 0;`
			`y = 1;`
			`arr[1] = 1;`
			`arr[2] = 2;`
			`arr[3] = 3;`

Figure 4.7: Examples of how reordering stores can potentially break dependencies.

Loop unrolling This pass performs loop unrolling, which expands the loop body across multiple iterations, reducing the overhead of checking the loop condition and updating the trip count, and expose further optimization opportunities (e.g., vectorization). Similar to `if/else` control dependencies, the loop body generally depends on the loop condition, and thus a full unrolling (i.e., expanding the loop body completely) can potentially break dependencies. Hence, we modify this pass such that it does not statically reason about the trip count of a loop and fully unroll the loop when its trip count is not an explicit constant, as shown in Figure 4.8 (a). However, if the trip count of a loop is specified as a constant, as shown in Figure 4.8 (b), we allow full unrolling because the loop condition does not depend on any loads.

Backend passes that can break dependencies: Given a dependency-preserving LLVM IR, the LLVM backend generates object code by passing the IR through a sequence of backend passes, which can also potentially break dependencies in the following ways:

	Before transformation		After transformation
(a)	`for (int i = 0; i < (x*0 + 2); i++)` `  arr[i] = i;`	⇒	`arr[0] = 0;` `arr[1] = 1;`
(b)	`for (int i = 0; i < 2; i++)` `  arr[i] = i;`	⇒	`arr[0] = 0;` `arr[1] = 1;`

Figure 4.8: Examples of loop unrolling. (a) statically computing the trip count and unrolling the loop potentially breaks control dependencies; (b) unrolling loops with explicit constant trip count does not break dependencies.

1. Data dependencies: The major backend pass that breaks data dependencies is the SelectionDAG-based instruction selection pass. To generate machine code, the LLVM backend first builds a per basic block structure called a selection DAG, which is a directed acyclic graph that represents the order of instruction within a basic block. It then goes through several rounds of node combining, which effectively performs a form of algebraic simplification, common subexpression elimination, constant folding, and strength reduction, etc. Similar to the modifications we made to the *instcombine* pass, we disable algebraic simplifications that can break dependencies.

2. Control dependencies: In addition to the IR-level control flow simplification, the LLVM backend can further simplify control dependencies, e.g., merging branches and eliminating empty blocks, including those conditional branches on which we potentially rely to preserve dependencies. We modify the code generation preparation (*codegenprepare*) pass and completely disable the control flow optimizer (*branchfolding*) pass to preserve control dependencies.

Chapter 5

Load-Store-Order-Preserving Compiler

This chapter describes the implementation of our approach to preserving atomic load-store ordering in the LLVM compiler for AArch64 targets. In LLVM IR, atomic load/store operations are special load/store operations with memory ordering parameters similar to their C/C++ counterparts (e.g., `memory_order_acquire`), and atomic read-modify-write operations (e.g., `compare_exchange_strong` and `fetch_add`) are represented as `atomicrmw` or `cmpxchg`. Similar to the dependency-preserving approach, we need to ensure that both IR-level optimizations and the backend generate code that preserves load-store ordering for atomic operations.

5.1 Target-Independent Optimizations

For IR optimization passes, we only need to focus on those passes that can potentially perform load-store reordering to atomic load/store operations, which fortunately is a small

subset of the IR optimization passes. For example, these passes include the loop invariant code motion pass (*licm*), the memory copy optimization pass (*memcpyopt*), the dead store elimination pass (*dse*), the SLP vectorization pass, etc. We carefully reviewed these passes and found that they do not perform load-store reordering for atomic operations by design. The reasons include: (1) the semantics of atomic operations disallow the optimization (e.g., atomic operations cannot be optimized into a memcpy/memset operation since it can potentially reorder the atomic operations and change the visible side effect) or (2) it is tricky to reason about the correctness of optimizations of atomic operations, and the optimization is not especially important in most cases. For example, the *licm* pass will optimize normal loads/store out of a loop but is conservative with atomic operations. As a result, we can enable all the original IR-level passes.

5.2 Backend Optimizations for AArch64

5.2.1 LLVM AArch64 Backend for C/C++ Atomics

Figure 5.1 shows how the LLVM backend compiles C++ atomics to assembly for AArch64 targets. In example (a), an atomic load/store with `memory_order_relaxed` ordering parameter is compiled to a normal load/store instruction, while an atomic load with `memory_order_acquire`[1] or store with `memory_order_release` is compiled to a load-acquire (`ldar`) or store-release (`stlr`), respectively, which are load and store instructions in AArch64 with implicit one-way barrier semantics. For example, the normal load and store in line 2 and 3 can be reordered by the processor at runtime, while the `ldar` in line 1 guarantees that loads/stores after it cannot be reordered before the load. Similarly, all loads/stores before

[1] `memory_order_consume` is not broadly supported by compilers due to challenges associated with preserving data dependencies. LLVM effectively converts it to the stronger ordering parameter `memory_order_acquire`.

the `stlr` in line 5 cannot be reordered after it. In example (b), a `fetch_add` operation is compiled to a loop that continuously attempts to atomically fetch and add one to the memory location. It takes advantage of the exclusive load/store (`ldxr`/`stxr`) pair, which has exclusive locking semantics on the load/store address to guarantee the read-modify-write is atomic. It is important to note that in the ARMv8 architecture, the success bit (`w10` in this example) of a successful store-exclusive is not supposed to introduce any dependency from the load-exclusive it is paired with (Pulte et al., 2018). As a result, the store in line 6 does not have a dependency on the load in line 2 and thus can be reordered before it. In example (c), a compare-and-swap operation is compiled to a conditional branch that compares the load value with the expected value and then decides whether or not it should store the new value to the memory. Since there is a control dependency (line 3) from the load part (line 2) to any stores after the compare-and-swap operation, those subsequent stores cannot be reordered before the load part. The conclusions from these three examples are (1) that atomic loads that have a stronger ordering parameter than `memory_order_relaxed` and atomic compare-and-swap operations already have an ordering constraint relative to subsequent stores, and (2) that we only need to preserve the ordering from the relaxed load and those `fetch_add`-like read-modify-write operations (with ordering parameters that do not have acquire semantics) to subsequent stores.

5.2.2 Forbidding Reordering of Loads and Stores in AArch64

To forbid a normal load from being reordered with subsequent stores in AArch64 targets, Boehm and Demsky (2014) propose that one could add either a fence or a bogus conditional branch after the load (i.e., adding control dependency from the load to subsequent stores). For cases in which a load is followed by a store, one could alternatively add a bogus address dependency (Maranget et al., 2012) from the load to the store to guarantee the ordering. Also, a similar strategy with respect to adding address dependency to insert (between the

	C++ code		AArch64 assembly

(a)	`r1 = arr[0].load(acquire);`
	`r2 = arr[1].load(relaxed);`
	`arr[2].store(0, relaxed);`
	`arr[3].store(0, release);`
		⇒	`1: ldar w1, [x8]`
			`2: ldr w2, [x8, #4]`
			`3: str wzr, [x8, #8]`
			`4: add x8, x8, #12`
			`5: stlr wzr, [x8]`

(b)	`r1 = arr[0].fetch_add(1, relaxed);`
	`arr[1].store(0, relaxed);`
		⇒	`1:.BB_1:`
			`2: ldxr w9, [x8]`
			`3: add w9, w9, #1`
			`4: stxr w10, w9, [x8]`
			`5: cbnz w10, .BB_1`
			`6: str wzr, [x8, #4]`

(c)	`int expected = 0;`
	`r1 = arr[0].compare_exchange_weak(`
	`expected, 1, relaxed, relaxed);`
	`return;`
		⇒	`1:.BB_1:`
			`2: ldxr w9, [x8]`
			`3: cbz w9, .BB_2`
			`4: clrex`
			`5: ret`
			`6:.BB_2:`
			`7: orr w9, wzr, #0x1`
			`8: stxr w10, w9, [x8]`
			`9: ret`

Figure 5.1: Examples of how LLVM backend compiles C++ atomic operations to assembly code for AArch64 targets. In each example, variable `arr` is an array of `atomic_int`, and register `x8` contains the base address of array `arr`.

target relaxed load and subsequent stores) a bogus load whose address depends on the target relaxed load. To better understand the performance characteristics of these options, we use micro-benchmarks written in assembly to benchmark the performance overhead of these options on an ARM Cortex-A72 core.

The first option is to simply replace the normal load with a `ldar` load, which has implicit acquire semantics. The second option is to replace the normal store with a `stlr` store, which has implicit release semantics. The third option is to insert a "`dmb ld`" fence before a relaxed store so that it waits for previous loads to finish. The fourth option is to add a bogus conditional branch after the normal load such that the branch condition uses the result of the load. Note that at the time of writing this dissertation, there is still some

uncertainty about what the branch target should look like. In this dissertation, we evaluate this strategy based on Pulte et al. (2018)'s model, in which any instruction succeeding a conditional branch in program order has control dependency on the loads that the branch has data dependency on. We will discuss this issue in more details in Section 6.2.4. The fifth option is to add a bogus load whose address depends on the target relaxed load. The sixth option is to add extra address/control dependency from the normal load to an existing subsequent store or conditional branch instruction. Figure 5.2 shows the performance overhead of these strategies, which is normalized to the performance of the micro-benchmarks without any load-store ordering constraints. The "Store" column represents the scenario in which a load is followed by a store, and the "Conditional Branch" column represents the scenario in which a load is followed by a conditional branch. This result shows that using release stores is the most expensive option and adding bogus conditional branches after relaxed loads is the least expensive option in either scenario for the processor we used, and that adding fences (i.e., the first three options) is more expensive than the other three alternatives. Given this preliminary result, we adopt the strategy of adding bogus conditional branch in the implementation of our load-store-order-preserving compiler. It is important to note that compared to adding bogus conditional branches, the strategies of adding address dependencies to existing stores/branches or inserting bogus dependent loads can be potential solutions for those processors that incur higher overheads from fake conditional branches.

Strategy/Subsequent Instruction	Store	Conditional Branch
Acquire Load	500.1%	267.7%
Release Store	1095.1%	382.0%
DMB LD Fence	457.1%	238.3%
Bogus Conditional Branch	28.6%	26.2%
Bogus Load	50.0%	26.4%
Extra Dependencies to Existing Store/Branch	50.0%	28.8%

Figure 5.2: Performance overhead incurred by different strategies of forbidding load-store reordering for micro-benchmarks.

Figure 5.3 (a) and (b) show examples in which a relaxed load is followed by an existing subsequent store or conditional branch in the same basic block, respectively. In both exam-

	C++ code		AArch64 assembly

(a)	`r1 = arr[1].load(relaxed);`
	`arr[0].store(0, relaxed);`
		⇒	`1: ldr w1, [x8, #4]`
			`2: and w2, w1, wzr`
			`3: cbnz w2, .BB_1`
			`4:.BB_1:`
			`5: str wzr, [x8]`

(b)	`r1 = arr[1].load(relaxed);`
	`if (r2)`
	`arr[0].store(0, relaxed);`
		⇒	`1: ldr w1, [x8, #4]`
			`2: and w9, w1, wzr`
			`3: cbnz w9, .BB_1`
			`4:.BB_1:`
			`5: cbz w2, .BB_2`
			`6: str wzr, [x8]`
			`7:.BB_2:`

(c)	`r1 = arr[1].load(relaxed);`
	`if (r1)`
	`arr[0].store(0, relaxed);`
		⇒	`1: ldr w1, [x8, #4]`
			`2: cbz w1, .BB_1`
			`3: str wzr, [x8]`
			`4:.BB_1:`

(d)	`r1 = arr[1].load(relaxed);` `arr[r1].store(0, relaxed);`	⇒	`1: ldr w1, [x8, #4]` `2: str wzr, [x8, w1, sxtw #2]`

Figure 5.3: Our approach to imposing the ordering between relaxed loads and subsequent stores. Register `x8` contains the base address of array `arr`. Bogus conditional branches are added intentionally to impose the load-store ordering in example (a) and (b), and example (c) and (d) do not require such extra ordering constraints because the ordering constraints exist in the source code inherently.

ples, we intentionally add a bogus conditional branch that uses the result of the load, i.e., lines 2 to 4 in Figure 5.3 (a) and lines 2 to 4 in Figure 5.3 (b). This intentional control dependency forces stores after the load to be visible after the load. Note that we add an `AND` instruction (specifically `AND` with `zero`) in lines 2 of both examples (a) and (b) to ensure that the conditional branch consistently takes the same direction to avoid too many branch mispredictions.

Another important observation is that for some relaxed loads, there already exist reordering constraints from the load to subsequent stores. For example, in the source code in Figure 5.3 (c), the conditional branch after the relaxed load already depends on the result of the load, so any stores after the load in the assembly naturally have a control dependency on the load

and must be visible after it without adding any redundant instructions. Similarly, Figure 5.3 (d) shows an example in which a subsequent store naturally has an address dependency on the load and thus we do not need to add extra reordering constraints. In order to optimize for these cases to avoid unnecessary overheads, we implement a local analysis that conservatively checks whether the address of a subsequent store or the condition of a subsequent branch depends on specific loads and use the analysis result to decide whether we need to add a bogus conditional branch after the loads.

To implement our solution, we made two modifications to the LLVM AArch64 backend:

1. Add extra ordering constraints for relaxed loads: We modify the code generation preparation pass such that before LLVM lowers optimized IR to machine code, it collects the set of relaxed loads that need extra ordering constraints (e.g., examples shown in Figure 5.3 (a) and (b)). We then intentionally add bogus conditional branches after the collected relaxed loads. Note that when there are multiple relaxed loads in a sequence, we only insert one bogus conditional branch whose condition uses the result of all those loads.

2. Preserving redundant data/control dependencies: After the above modification to the code generation preparation pass, we still need to ensure that later backend passes (e.g., the SelectionDAG-based instruction selection and control flow optimization pass) do not optimize these instructions away, e.g., eliminating "`and w2, w1, wzr`".

Chapter 6

Evaluation

In this chapter, we evaluate the cost of the two approaches to avoiding out-of-thin-air behaviors in C/C++ for AArch64 targets. In our evaluation, we report execution times on a Firefly-RK3399 board, which has a six-core 64-bit CPU (two ARM Cortex-A72 cores and four ARM Cortex-A53 cores), 4 GB memory, and runs Ubuntu 16.04.2. We have made both our compiler implementations and benchmarks publicly available at `http://plrg.eecs.uci.edu/oota-html`. As Sullivan (2017) shows, the performance results can vary depending on the processor in question. More specifically, their results seem to suggest that dependencies and fences may exhibit less performance penalty in ARMv8 than in ARMv7 and Power architectures. Ideally, the evaluation would have been more complete if we also considered the ARMv7 and Power architectures; however, in LLVM, they use different backends (which does require review and modifications) to generate architecture-specific code, so this is a non-trivial effort. As a first step, we believe that evaluating the approaches on a relatively new version of 64-bit ARM processor could still be a useful indicator for future processors, and decided to leave the evaluation on the ARMv7 and Power architectures as future work.

6.1 Cost of Preserving Dependencies

Although avoiding out-of-thin-air behaviors applies only to multi-threaded code, our dependency-preserving approach incurs overhead for both single-threaded and multi-threaded programs. Single-threaded code represents a worse case scenario—the memory system bandwidth is not utilized by other cores and thus the extra instructions we add have a relatively higher cost. Thus, we measure the overheads of our dependency-preserving optimizations on single-threaded code.

6.1.1 Single-Threaded Programs

We ran each C/C++ benchmark in SPEC CPU2006 (Henning, 2006) under four compiler configurations. The configuration "Full Optimizations" is the stock LLVM compiler with all optimizations enabled (`-O3`); the configuration "No Optimization" is the stock LLVM compiler with all optimizations disabled (`-O0`); the configuration "Dependency-preserving" is our dependency-preserving compiler. Due to the amount of engineering work needed to review/modify each optimization pass, we only select a core set of IR-level optimization passes (35 out of 46) to carefully review and modify when necessary to implement the dependency-preserving compiler. The configuration "Partial Optimization" is the stock LLVM compiler whose IR-level optimizations only include the same core set of passes that are enabled in our dependency-preserving compiler.

Note that the "No Optimization" configuration (`-O0`) naturally preserves dependencies; however, the benchmarks under this configuration execute with an average (geometric mean) slowdown of 155.9% and a maximum slowdown of 580.9%. Figure 6.1 shows more detailed performance overhead of each benchmark under configurations "Partial Optimization" and "Dependency Preserving", with each normalized to the performance under "Full Optimiza-

tion" (`-O3`) configuration. Under the "Partial Optimization" configuration, the benchmarks incur an average of 1.8% slowdown and a maximum of 11.6% slowdown. Our dependency-preserving compiler has an average 3.1% slowdown and a maximum of 17.6% slowdown. Given the fact that we completely turn off 11 IR-level passes in our dependency-preserving compiler, which roughly accounts for the 1.8% overhead as shown under the "Partial Optimization" configuration, it is likely that one could further reduce the overhead of preserving dependencies by analyzing those optimization passes. There also remain opportunities for further optimization of the passes that we modified for the dependency-preserving memory model.

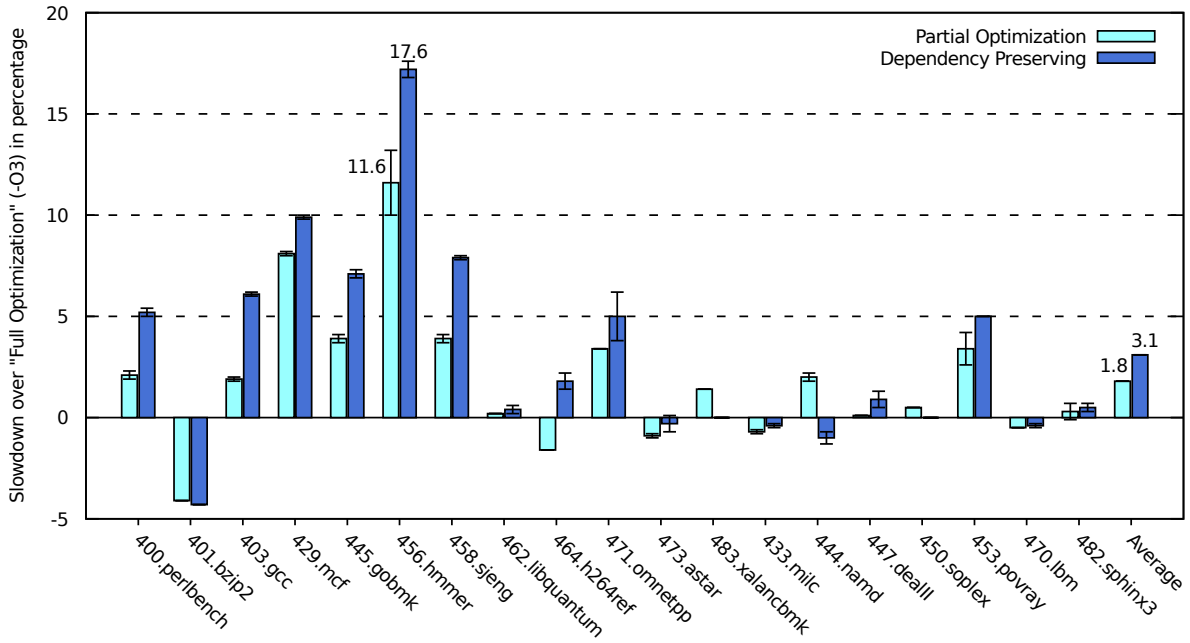

Figure 6.1: Performance overhead (in percentage) introduced by different compiler configurations compared to the full optimization configuration (`-O3`) for C/C++ benchmarks in SPEC CPU2006.

Speedup in Single-Threaded Runs As shown in Figure 6.1, we observe speedup in the single-threaded runs for some benchmarks under our dependency-preserving compiler. A possible explanation is the non-linear interaction between some optimization passes on

some benchmarks. In fact, researchers have shown that different compiler options or transformation combination could have a significant impact on performance factors (e.g., cache accesses) (Pan and Eigenmann, 2006; Cavazos et al., 2007). One supportive observation in our case is that some benchmarks such as "401.bzip2" under the "Partial Optimization" configuration also have a speedup over the baseline "Full Optimization" configuration. Moreover, our dependency-preserving compiler does require modifying some backend passes such that they do not eliminate intentionally added `AND` instructions or conditional branches; and we have observed that simply disabling the backend control flow optimizer pass (i.e., *Branch-Folding*) in the stock LLVM ("`-O3`") yields speedups for some single-threaded benchmarks. To give a more detailed comparison, we also list the overhead incurred by our dependency-preserving compiler over the "Partial Optimization" configuration in Figure 6.2.

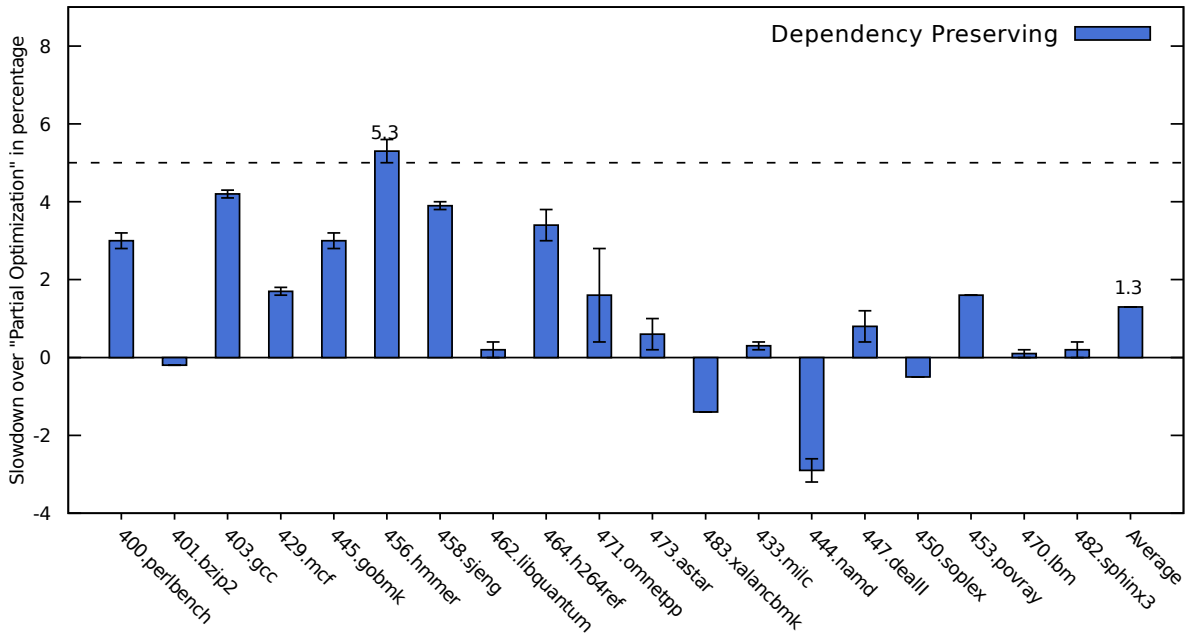

Figure 6.2: Performance overhead (in percentage) introduced by our dependency-preserving compiler compared to the "Partial Optimization" configuration for C/C++ benchmarks in SPEC CPU2006.

6.1.2 Multiple Copies of Single-Threaded Programs

To evaluate the performance overhead in a multi-core environment, we ran two copies of each C/C++ benchmark in SPEC CPU2006 at the same time, with each copy running on a Cortex-A72 core. We report the performance overhead of our dependency-preserving compiler compared to the stock LLVM with all optimizations enabled (`-O3`) in Figure 6.3. In this scenario, our dependency-preserving approach incurred an average slowdown of 2.6% and a maximum slowdown of 13.7%, which is smaller than that of running in a single-copy scenario in Section 6.1.1. A likely explanation is that our approach to preserving dependencies increases the number of instructions that are executed but does not significantly increase the number of memory accesses to data. As a result, when we run multiple copies of the same program simultaneously, the memory bandwidth that is accessible to each copy is reduced, and hence the cost of running the extra CPU instructions becomes relatively smaller (especially for memory-bounded programs). This experimental result indicates that in multi-core environments in which more than one core is used, the performance overhead incurred by our dependency-preserving compiler is likely to be smaller than that incurred in the single-core scenario.

6.2 Cost of Forbidding Load-Store Reordering

Unlike the dependency-preserving approach, forbidding load-store reordering to avoid out-of-thin-air behaviors only affects relaxed atomics in C/C++11. Hence, for example, the load-store-order-preserving approach should impose no overhead on the SPEC CPU2006 benchmarks because they do not use any C/C++ relaxed atomics. We believe that relaxed atomics will primarily appear in concurrent data structure code, while most other program code would not be affected since they would likely use other primitives that provide stronger semantics, e.g., locks and atomics with `memory_order_seq_cst`. Hence, we

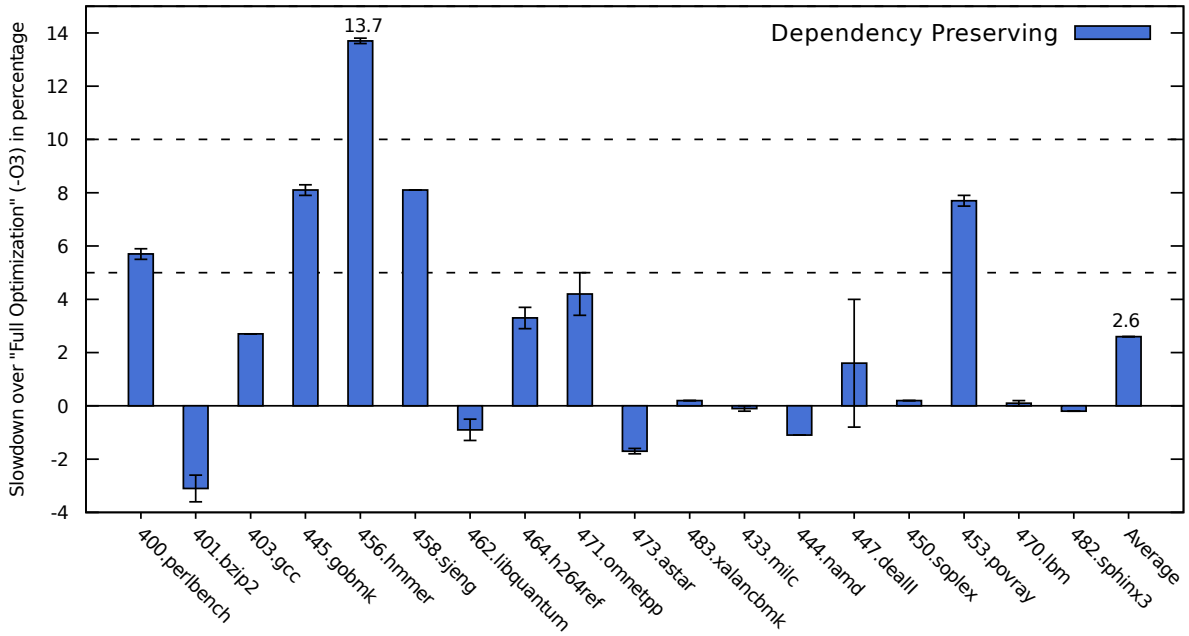

Figure 6.3: Performance overhead (in percentage) introduced by our dependency-preserving compiler compared to the full optimization configuration (`-O3`) for C/C++ benchmarks in SPEC CPU2006 with two copies of each benchmark running at two cores simultaneously. We omit the "429.mcf" benchmark here because running two copies at the same time requires more than 4 GB memory and thus causes out-of-memory error.

focus on evaluating the performance overhead incurred by forbidding load-store reordering for real-world concurrent data structures. The results can be roughly viewed as the upper bound of the performance overhead of this approach. The performance impact on full applications would depend on how much time those applications spend in concurrent data structure code. Ideally, we would also like to benchmark full applications; however, many existing multi-threaded applications that we have access to are not kept up-to-date with the C/C++ memory model, and porting them to use C/C++ atomics is a non-trivial effort. Hence we leave it as future work.

6.2.1 Concurrent Data Structures with Multiple Threads

In this evaluation, we gather a total set of 43 real-world concurrent data structures from several different sources, which range from basic synchronization primitive implementations, concurrent queues/stacks/deques to concurrent maps. Most of these data structures are lock-free, and all of them intensively utilize C/C++11 atomics. In more details, among these concurrent data structures, we collect 18 of them from the CDS C++ library (Khiszinsky, 2017), 13 of them from the Folly library (Facebook, 2018), 4 different implementations of concurrent maps from the Junction library (Preshing, 2018), 2 queue implementations by Rigtorp (2017a,b), and 6 benchmarks used in CDSSpec (Ou and Demsky, 2017).

We ran each benchmark using 7 compiler configurations: (1) the stock LLVM compiler with all optimizations enabled (`-O3`), i.e., *Full Optimization*; (2) our load-store-order-preserving compiler which adds bogus conditional branches after relaxed loads (*Bogus Conditional Branch*); (3) a variant of configuration 2 which adds address dependencies to existing stores rather than bogus conditional branches if there is a subsequent store after a relaxed load (*Address Dependency to Store*); (4) a modified compiler which adds address dependencies from relaxed loads to a subsequent load, which can be an existing load if any or an intentionally inserted bogus load otherwise (*Bogus Load*). Note that for a target relaxed load, we insert a bogus load whose address is the same as the relaxed load to avoid cache misses; (5) a modified compiler which treats relaxed loads as acquire loads (*Acquire Load*); (6) a modified compiler which treats relaxed stores as release stores (*Release Store*); and (7) a modified compiler which inserts "`dmb ld`" fences before relaxed stores (*DMB Fence*).

Since the Firefly-RK3399 board has two faster ARM Cortex-A72 cores and four slower ARM Cortex-A53 cores, it can potentially increase the noise in our performance evaluation if we run the benchmarks with multiple threads across the two different types of cores. Hence, we ran each of our benchmark with two threads, and each thread exclusively runs on a Cortex-

A72 core. We ran each benchmark test case for 5 times and use the average (arithmetic mean) of those 5 runs as the execution time for each benchmark test case. For a benchmark in which there exist multiple variants, we use the geometric mean of the execution time of all the variants as the execution time of that benchmark. Although we ran the benchmarks with only two threads in this experiment, it is important to note that the extra overhead (i.e., extra dependencies or fences) that we introduce in this approach is local to each core and thus should not result in extra communication between cores, and hence one would not expect scaling issues; moreover, if the processor has limited memory bandwidth, as the number of cores utilized increases, the relative overhead of these extra dependencies or fences should become smaller.

	Multiple Threads		Single Thread	
Configurations/Overheads	Average	Maximum	Average	Maximum
Bogus Conditional Branch	-0.3%	6.3%	-0.0%	5.2%
Address Dependencies to Store	1.3%	23.2%	0.5%	8.7%
Bogus Load	2.6%	42.9%	2.8%	14.7%
Acquire Load	0.4%	27.5%	2.1%	42.7%
Release Store	3.6%	82.6%	6.8%	38.9%
DMB Fence	-0.1%	32.0%	3.2%	25.9%

Figure 6.4: Performance overheads (over full optimizations) incurred by different strategies of forbidding load-store reordering for concurrent data structure benchmarks on Cortex-A72 cores. The "Multiple Threads" columns show results for benchmarks running with two threads, and the "Single Thread" columns show results for benchmarks running with a single thread.

The performance overheads of different strategies to preserve load-store ordering (running with two threads) over the performance under full optimizations are shown in the "Multiple Threads" columns in Figure 6.4. The "Average" column shows the geometric mean of the execution time of our benchmarks, and the "Maximum" column shows the maximum overhead incurred by the corresponding strategy. We can see that the "*Bogus Conditional Branch*", "*Acquire Load*" and "*DMB Fence*" strategies incur an average overhead of less than 0.5% across the 43 benchmarks on average. Notably, the "*Bogus Conditional Branch*" strategy does not incur an overhead on average and only incurs a maximum of 6.3% overhead. All other strategies have higher maximum overhead than the "*Bogus Conditional Branch*"

strategy, indicating that they are less desirable approaches to preserving load-store ordering in our experimental setting. We also show the performance results of each benchmark for the *Bogus Conditional Branch* strategy in Appendix B.

Contention Under this experimental setting, we found that some of our benchmarks have a faster execution time under our load-store-order-preserving compilers over full optimizations (`-O3`), such as the Folly UnorderedAtomicInsertMap implementation (Facebook, 2018). A possible explanation is that there exists contention in the data structure, and adding extra instructions to implement ordering constraints alleviates this contention. To better compare the performance of our approach, we also run our benchmark in a single-threaded (contention-free) setting in Section 6.2.2.

6.2.2 Concurrent Data Structures with a Single Thread

We run our benchmarks in a single thread on an ARM Cortex-A72 core in order to study the performance overhead of our approach without the contention issue under the 7 compiler configurations described in Section 6.2.1. For example, for a concurrent queue, we ran the queue with a single thread, which executes a certain number of `enqueue` method calls and then a certain number of `dequeue` method calls. The results are shown in the "Single Thread" columns in Figure 6.4. We can see that without contention, the "*Bogus Conditional Branch*" strategy does not incur an overhead over full optimizations on average and only incurs a maximum of 5.2% overhead. It also shows that the "*Bogus Load*", "*Acquire Load*", "*Release Store*" and "*DMB Fence*" strategies are more expensive on average and in worst case, which agrees with our micro-benchmarking results shown in Figure 5.2. Notably, even though the "*Bogus Load*" strategy only adds address dependencies to existing or bogus loads (which may seem inexpensive), it is still not desirable relative to the "*Bogus Conditional Branch*" strategy. A possible explanation is that the added address dependencies can halt

the execution of all future memory operations.

Thus, among the six strategies we implemented, when we consider both the multi-threaded and single-threaded experiment results, the "*Bogus Conditional Branch*" strategy is the most desirable under the processor we use because it has the lowest average overhead and worst-case overhead in both the multi-threaded and single-threaded runs compared to all other strategies. Also, if we consider the single-threaded runs, the "*Address Dependencies to Store*" strategy is only slightly less desirable than the "*Bogus Conditional Branch*" strategy, and it may serve as a potential approach for processors that incur higher overheads from fake conditional branches.

6.2.3 Optimizations for the Load-Store-Order-Preserving Approach

There are two core ideas behind our optimizations to alleviate the performance overhead of enforcing the load-store ordering. One is to take advantage of existing ordering constraints that are intrinsic to the source code to avoid adding unnecessary extra ordering constraints, and the other one is to move the added ordering constraints out of the critical sections when possible. We discuss them in more details as follows.

Avoid Unnecessary Ordering Constraints As shown in Figure 5.3 (c) and (d), a relaxed load can automatically have ordering constraints to subsequent stores because of existing control or address dependencies, e.g., when the result of the load is used to compute the condition of an immediately following conditional branch. Another scenario to optimize is when a relaxed load is followed by `fetch_add`-like atomic operations with acquire and release semantics and there does not exist any atomic store in between. One notable real example is the synchronizing barrier implementation (Velikov, 2012), with an interesting code snippet shown in Figure 6.5. The `fetch_add` operation that immediately follows the

relaxed load on variable `step_` has the `memory_order_acq_rel` memory order, and the LLVM backend will transform this `fetch_add` operation to acquire load-exclusive and release store-exclusive instructions. As a result, the `fetch_add` operation effectively acts as a fence that forbids the relaxed load and any subsequent stores to be reordered across it. Another such pattern is a relaxed load immediately followed by a `CAS` operation. In the implementation of our load-store-order-preserving compiler, we have an analysis to identify these patterns for relaxed loads to avoid adding unnecessary ordering constraints.

```
unsigned step = step_.load(relaxed);
if (nwait_.fetch_add(1, memory_order_acq_rel) == n_ - 1) {
 // Subsequent stores...
}
// Other subsequent stores...
```

Figure 6.5: A relaxed load followed by a `fetch_add`-like read-modify-write operation (no other atomic store in between) is naturally guaranteed to be ordered before subsequent stores after the read-modify-write operation.

Move Added Ordering Constraints out of Critical Sections If a relaxed load in the critical path requires adding extra ordering constraints, we can potentially reduce the penalty if we can safely move the intentionally added ordering constraints out of the critical path. Figure 6.6 shows the code for the `unlock` method of the Ticket Lock (Reed and Kanodia, 1979) implementation. This lock data structure maintains a `turn` variable to indicate whose turn it is to take the lock. For a thread that holds the lock, the `unlock` method simply increments the `turn` variable to allow the next waiting thread to acquire the lock. Since the thread holding the lock has exclusive access to the `turn` variable, it does not use an atomic `fetch_add` operation, which is potentially more costly than a plain load and store. In this case, to ensure that the relaxed load is ordered before all subsequent stores, a straightforward approach is to add a bogus conditional branch right after the relaxed load and before the release store; however, this intentionally added control dependency is in the critical section because the `turn` variable has not been incremented yet (so the waiting thread potentially needs to wait for a longer time). One observation is that the store in this case has release

semantics, meaning that the load cannot be reordered across it. Hence, we can instead add the bogus conditional branch after the release store. As a result, the critical section does not contain any added control dependency that delays the process of releasing the lock. In our implementation, for a relaxed load, we try to find the latest release store in the same basic block of that load and add the bogus conditional branch after that last release store.

```
void unlock() {
 unsigned my_turn = turn.load(std::memory_order_relaxed);
 // Still in the critical section
 turn.store(my_turn + 1, std::memory_order_release);
 // Not in the critical section anymore
}
```

Figure 6.6: Example of a relaxed load followed by a release store. Since the relaxed load cannot be reordered across the release store, we can safely delay adding a bogus conditional branch till after the release store rather than before the release store.

6.2.4 Subtleties of Control Dependencies in the ARMv8 Memory Model

According to the formalized ARMv8 memory model by Pulte et al. (2018), the definition of control dependency is straightforward: all instructions that are after a conditional branch in program order are control-dependent on the loads that the branch is data-dependent on. This notion of control dependency is syntactic and holds regardless of the instructions in the untaken branch, and without any distinction of true and false dependencies[1]. The implementation of our load-store-order-preserving compiler is essentially based on this model, and the bogus conditional branch that we intentionally added directly jumps to the next instruction, i.e., the taken and untaken branches are at the same location. Take Figure 6.7 as an example, in which we load a value to register `w1` (line 1), perform an `AND` operation on

[1]The ARMv8 architecture reference manual defines true dependencies and false dependencies, which essentially aims at requiring the processor implementation to preserve only true dependencies (but not false dependencies). For example, for instruction "`and x2, x1, xzr`", register `x2` has false dependency on register `x1`, while for instruction "`and x2, x1, x0`", register `x2` has true dependencies on registers `x0` and `x1`. However, it is still not clear how one could precisely define this in a satisfactory way.

the load result with zero (line 2), branch to the next instruction if the value in register `w2` is not zero (line 3), and then perform a store right after the branch(line 5). Here, the store in line 5 has control dependency on the load in line 1 regardless of the fact that line 2 has false data dependency and the conditional branch in line 3 jumps directly to the next instruction (i.e., line 5).

```
1:  ldr w1, [x0]
2:  and w2, w1, wzr
3:  cbnz w2, .BB_1
4:.BB_1:
5:  str wzr, [x3]
6:  ret
```

Figure 6.7: According to Pulte et al. (2018)'s memory model, the store in line 5 has control dependency on the load in line 1.

However, the precise definition of control dependency is not entirely clear in the text of the ARMv8 architecture reference manual (ARM, 2016). As it states:

> "A Control dependency from a read R_1 to a subsequent instruction I_2 exists if and only if there is a Register dependency from the data value returned by R_1 to the data value used in the evaluation of a conditional branch or the determination of a synchronous exception on an instruction and I_2 is only executed as a result of one of the possible outcomes of that conditional branch or synchronous exception."

The problematic phrase is that "I_2 is only executed as a result of one of the possible outcomes of that conditional branch", which by some way of interpretation could mean that the store in line 5 in Figure 6.7 is not control-dependent on the load in line 1 because the store executes in all possible (taken or untaken) branches regardless of the branch condition. At the time of the writing of this dissertation, there also exist rumors that there is research effort in formalizing a weaker definition of control dependency in the ARMv8 memory model than that defined in Pulte et al. (2018)'s model. However, it is not clear what such a

definition would become, and the exact placement of the branch target in our load-store-order-preserving compiler would also remain unclear till a precise definition is known. In Figure 6.8, we hypothesize a stronger strategy to enforce load-store ordering, which hopefully would survive future weakening to the notion of control dependency. In this strategy, instead of letting the added conditional branch (line 3) jump right to the next instruction, we let it jump to an infinite loop (line 6 and 7). We believe this is a strong constraint that would form control dependency because at the point of executing the conditional branch, it can potentially execute the next store or jump to execute an instruction that halts the whole program, although at runtime the branch of the infinite loop would never be taken.

```
1:  ldr w1, [x0]
2:  and w2, w1, wzr
3:  cbnz w2, .BB_2
4:  str wzr, [x3]
5:  ret
6:.BB_2:
7:  b .BB_2
```

Figure 6.8: In this example, the target of the conditional branch in line 3 is an infinite loop (line 6 and 7), instead of the next instruction as shown in Figure 6.7. This does not change the semantics of the original code since the branch would not be taken at runtime.

Following this hypothesis, we implement a variant of our load-store-order-preserving compiler, which we refer to as the *strong-load-store-order-preserving* compiler. It differs from our load-store-order-preserving compiler in the following ways: (1) for a relaxed load that requires adding bogus control dependencies, we insert a bogus conditional branch that jumps to an infinite loop rather than to the immediately subsequent instruction; (2) we turn off the branch analysis, i.e., we add bogus control dependencies even for relaxed loads whose results are used in a subsequent conditional branch (as shown in Figure 5.3 (c)); and (3) similar to `fetch_add` like operations, we treat non-acquire compare-and-swap operations as regular relaxed loads and adds bogus control dependencies for their load parts.

We then run the same set of concurrent data structure experiments as shown in Section 6.2.2

and Section 6.2.1 under our strong-load-store-order-preserving compiler on the Cortex-A72 cores. For the multi-threaded runs, our strong-load-store-order-preserving compiler incurs an average of 0.6% overhead and a maximum of 22.7% overhead over baseline (`O3`); for the single-threaded runs, it incurs an average of 1.0% overhead and a maximum of 9.5% overhead over baseline. Compared to the results of our load-store-order-preserving compiler (for Cortex-A72 cores) shown in Figure 6.4, we can see that our strong-load-store-order-preserving compiler does incur slightly higher overhead than our load-store-order-preserving compiler (for both single-threaded and multi-threaded runs). A likely explanation for this is that our strong-load-store-order-preserving compiler treats more operations as relaxed loads (and hence adds more bogus control dependencies than the load-store-order-preserving compiler), and that adding branches that jump to infinite loops (rather than the immediately subsequent instructions) can increase the cost of branch mispredictions.

Chapter 7

Related Work

The out-of-thin-air problem has been shown to be a challenging issue that makes informal reasoning, formal reasoning, and compiler optimization very difficult (Boehm and Demsky, 2014; Batty et al., 2013; Vafeiadis et al., 2015; Vafeiadis and Narayan, 2013). In spite of much research on high-performance concurrent programming languages, we still do not have a definitive solution to the out-of-thin-air problem. Indeed, Batty et al. (2015b) show that there is no per-candidate-execution solution to the problem.

7.1 The Java and C/C++ Memory Models

The original Java memory model (Gosling et al., 1996) has been shown by Pugh (1999) to be flawed, which is both too strong to admit compiler optimizations and too weak for some commonly used programming idioms to work correctly. Various proposals for fixing the original model were found to either prohibit optimizations or allow undesired behaviors (Maessen et al., 2000; Manson and Pugh, 2001; Adve, 2004; Saraswat, 2004). Later, Manson et al. (2005) propose a new Java memory model (Shipilëv, 2016a,b), which disallows out-of-thin-air

executions by establishing a notion of justifying (*well-behaved*) executions: a write can only perform early when there exists a justifying execution for it. As a result, it disallows the canonical out-of-thin-air example shown in Figure 2.2 by showing that there cannot exist a justifying execution for the write of `42` to either variable `x` or `y`. However, this model was also found unsound with respect to some common compiler optimizations (Cenciarelli et al., 2007; Ševčík and Aspinall, 2008) such as redundant read elimination. This issue remains unresolved so far. The fact that our dependency-preserving approach supports normal memory accesses rather than just C/C++ atomics indicates that it is a promising direction to explore for the Java memory model.

Unlike the Java memory model, the C/C++11 memory model (Boehm and Adve, 2008; JTC, 2011; Becker, 2011; Batty et al., 2011) is shown to be sound with respect to compiler optimizations (Morisset et al., 2013), and researchers have used it in many aspects including compilation schemes to specific architectures, model checking, compositional library abstraction, program logics and concurrent data structure specifications (Batty et al., 2011, 2012; Sarkar et al., 2012; Norris and Demsky, 2013; Batty et al., 2013; Vafeiadis and Narayan, 2013; Turon et al., 2014; Ou and Demsky, 2017). However, it does not forbid out-of-thin-air executions; and the C++14 memory model (JTC, 2014) does not clearly define out-of-thin-air behaviors and only vaguely states that implementations should ensure that out-of-thin-air values that circularly depend on their computations should be disallowed. In our dependency-preserving approach, we formally define a notion of dependency and evaluate a prototype implementation on widely deployed commercial hardware.

Different from the axiomatic aspect of the C/C++11 memory model, Nienhuis et al. (2016) present an equivalent operational operational for it. To represent the relaxed behaviors allowed by the C/C++11 memory model, their semantics introduce symbolic steps, which allow a read to read a symbolic value that is determined later. Notably, they allow cyclic justification to account for out-of-thin-air executions.

7.2 Forbidding OOTA While Allowing Compiler Optimizations

Researchers have also proposed memory models whose goal is to disallow out-of-thin-air behaviors while embracing compiler optimizations. Essentially, these memory models and the two approaches we study in this dissertation are tradeoffs between the simplicity of the memory model and the degree of compiler optimizations allowed. For instance, these memory models primarily targets at forbidding the canonical OOTA example shown in Figure 2.2 and allowing the load buffering example shown in Figure 2.1. Our approaches take a different direction to impose some constraints on compiler optimizations, which forbids the executions in both examples; and we provide an initial study of the runtime overhead of restricting compiler optimizations to eliminate out-of-thin-air behaviors.

Event-structure-based game-like model for Java Jeffrey and Riely (2016) have proposed a weak memory model based on event structures, which are sets of memory access events with some causal order and conflict relationship. In their approach, a program is viewed as a single event structure, and an execution is a justified configuration of the event structure. To prohibit out-of-thin-air executions, they introduce the notion of *well-justified* configurations, and they show that well-justified configurations are sequentially consistent in the absence of data races. However, as the authors point out, their model does not allow the reordering of independent reads, which makes it too strong to be efficiently implemented on architectures such as Power and ARM. The authors suggest a fix for it, which, however, may invalidate some other guarantees their initial proposal provides.

Event-structure-based model for C/C++ Pichon-Pharabod and Sewell (2016) have also proposed an event-structure-based memory model with the goal of providing an en-

velope around compiler and processor optimizations and forbid out-of-thin-air executions. Their basic idea is to use an event structure to represent the current state for each thread and capture all of its potential executions, and to allow interleaving of threads and transformations over the event structure to abstract optimizations. They introduce mechanisms such as deordering, merging, execution steps, value range speculation steps, etc, to simulate the effect of compiler optimizations from thread-local to inter-thread optimizations. Their model considers locks, relaxed atomics, and ordinary memory accesses, and it was later pointed out that it does not validate some weak behavior allowed by the ARMv8 memory model (architecturally allowed at the time) and may not be compiled to ARM without adding fences (Kang et al., 2017).

Promising semantics Kang et al. (2017) propose a memory model that forbids out-of-thin-air behaviors based on operational semantics with *timestamps* and *promises*. The core idea is that a thread can "promise" a write (i.e., issue the write early) if the thread can fulfill that promise later without interacting with other threads (i.e., the promise is *thread-locally certified*), which simulates the effect of load-store reordering by allowing a store to be visible to other threads early. They attach a unique timestamp to each write, and each thread records the largest timestamps for each memory location that it has observed. To ensure cache coherence, a read can only from a write that has a timestamp that is at least as large as the one observed so far. Their model handles the C/C++ release/acquire and relaxed atomics, and ordinary memory accesses, and provides well-defined semantics for programs with data races. Because of the idea that a promise only needs to be locally certified, many thread-local optimizations are allowed by the model. They show their model is sound with respect to some basic reordering optimizations and can be compiled to x86 and Power using the expected compilation schemes, and Podkopaev et al. (2017) also show the proof of its compilation correctness to ARM.

Generative operational semantics for Java Jagadeesan et al. (2010) propose an operational memory model for Java with the goal of providing a generative view of the operational semantics to the Java memory model. For data-race-free programs, their model is consistent with the Java memory model; and when there are data races, their model would allow some optimizations that the Java memory model disallows, e.g., validating the roach-motel reordering and some peephole optimizations. To model compiler and processor optimizations, it introduces the notion of *speculation*. Speculative executions create two copies of the original process, the *initial* copy and the *final* copy, which are executed independently. The initial copy assumes nothing, and the final copy assumes the speculated writes. A valid execution is one where every speculation can be *finalized*, and when the speculation is finalized, only the final copy remains. To ensure the properties of no-thin-air-reads and data-race-freedom, it imposes constraints on the speculation such that it is *not self-justifying*, but is *initial, consistent and timely*.

Operational aspects for C/C++ Podkopaev et al. (2016) propose an operational memory model for C11 that supports a large subset of the features of the C11 memory model. Their model accounts for relaxed behaviors by introducing the notions of *viewfront* and *operation buffer* that has a nesting structure. Each thread maintains a viewfront to record which writes it has observed and can be used to ensure cache coherence. The per-thread operation buffers allow a thread to postpone the execution of an action and allow reads to return symbolic values, which is used to account for speculating relaxed behaviors. With nested buffers, they can account for optimizations such as merging writes that appear in both branches of an `if/else` block. It has not been proved that their model would support standard compiler optimizations.

Theory of memory models Saraswat et al. (2007) propose a model (for the X10 language) that represents a program as a graph, in which nodes represent memory actions and

edges represent program order or synchronization order. Relaxed behaviors are then represented through the notion of a *step*, which performs transformations on the graphs such as joining and splitting of the nodes. However, the model is built upon a restricted language and does not support general branches, and thus it is not clear how hard it would be to map their transformations to program transformations in a general-purpose language like Java.

Happens-before memory model Zhang and Feng (2013, 2016) propose an operational memory model that is based on a replay mechanism to simulate speculation and allow writes propagate to different threads at different times. Their model forbids some but not all OOTA behaviors. Notably, their replay mechanism must keep track of syntactic dependencies between instructions, and hence it would disallow some compiler optimizations.

It is important to note that the approaches that fall into this category generally expose memory models that are substantially different from (and arguably more complicated than) the C/C++ memory model, and it remains unclear whether they are sufficiently easy to understand for compiler writers and developers.

7.3 Case-Based Approaches

Some other approaches are to constrain the usage of atomics to specific cases and provide simple semantics for those cases. The well-known DRF-SC (Adve and Hill, 1990; Gharachorloo et al., 1992) model guarantees that all executions are sequentially consistent if there is no sequentially consistent execution with a data race, which is easy to understand and allows a wide range of optimizations (Ševčík, 2011; Morisset et al., 2013). Other work enumerates common use cases for relaxed atomics and provides semantics for those use cases (Sinclair et al., 2017). However, there are two challenges with these approaches: (1) memory model designers must ensure that the cases handled cover the important usage scenarios and (2)

bugs can produce behaviors that fall outside the well defined cases and then the memory model may provide little or no guarantees as to the program's behaviors.

7.4 Enforcing Stronger Memory Models

Preserving dependencies Boehm and Demsky (2014) have proposed the approach of preserving syntactic dependencies to forbid out-of-thin-air behaviors, yet they do not strictly define the notion of syntactic dependencies, and the overhead of preserving such dependencies was unknown before. McKenney et al. (2016) have proposed an approach based on preserving semantic dependencies rather than syntactic dependencies, which presumably allows more optimizations than the syntactical approach. However, it is unclear how one can precisely define such a notion of semantic dependencies.

Forbidding $sb \cup rf$ cycles Boehm and Demsky (2014) propose the approach of ensuring the relation $sb \cup rf$ is acyclic on relaxed atomics. This model forbids the load-buffering behavior that is allowed by the ARM and Power memory models and by some compiler optimizations that potentially reorder reads/writes to different locations. Hence, implementing this model would require restricting some compiler optimizations and introducing additional memory fences or dependencies, and the actual overhead was unclear. Vafeiadis and Narayan (2013) also consider this memory model and have shown the soundness of a program logic called *relaxed separation logic* (RSL) for C11 relaxed atomic accesses based on such a strengthened model. Our work complements this by providing an initial evaluation on the overhead of the approach for execution of real-world concurrent data structure code on a mainstream processor.

Local DRF Dolan et al. (2018) have recently proposed a memory model that provides a property called *local data race freedom*, which guarantees that all data-race-free portions of a program still have sequential consistency semantics. They show that to implement their memory model, one would need to preserve the ordering between loads and stores. They implement it on OCaml with similar strategies that we use in our load-store-order-preserving compiler and show that the average overhead over stock OCaml compiler for ARMv8 architecture with sequential programs is ∼0.6%. Although both their results and ours suggest that the overhead of preserving load-store ordering is relatively low on ARMv8 architecture, it is important to note the differences: (1) the primary goal of their model is to provide the local DRF property but not to prohibit OOTA behavior, although their model effectively disallows OOTA behavior; and (2) their results are based on OCaml. Similar to the Java memory model, the OCaml memory model (Dolan et al., 2014) specifies atomic and normal memory accesses, and atomic accesses are sequentially consistent. Their approach needs to preserve load-store ordering for all normal (non-atomic) accesses; while our approach targets the C/C++ memory model, which only affects the C/C++ atomics.

The SRA memory model Lahav et al. (2016) propose the SRA memory model, which imposes release-acquire semantics on all memory accesses and requires that the union of the (per-location) modification-order and happens-before relations be acyclic. This model effectively forbids OOTA behavior and can be efficiently implemented on the x86 architecture; however, its strong constraints are likely less acceptable on ARM and Power processors.

Enforcing TSO Ševčík et al. (2013); Demange et al. (2013) propose TSO for C and Java, which is strictly stronger than our approach of preserving load-store ordering and disallows OOTA behavior. While this approach can be reasonable for x86 multiprocessors, the overhead of these proposals has not been practically studied and may not be the desired approach when compiled for the ARM and Power processors, and it may not be viable

solution for languages that are intended to support portable high-performance concurrent programs.

Enforcing SC Researchers (Marino et al., 2011; Singh et al., 2012) have suggested stronger memory models, e.g., sequential consistency, in which out-of-thin-air behaviors are prohibited. They show that the cost is low when they implement such memory models on specialized hardware. Our approaches also explore stronger memory models than the existing C/C++ memory model; however, the constraints we impose in general are much weaker than the sequential consistency memory model. In addition, both of our approaches directly target existing widely-deployed commercial processor designs that implemented a relaxed memory model. Liu et al. (2017) have proposed a stronger Java memory model, which by default has sequential consistency semantics. They show that the overhead is arguably acceptable for server-side applications running on Intel x86 architectures.

Benchmarking weak memory models There is also work that benchmarks the performance of weak memory models. Ritson and Owens (2016) focus on investigating the cost of prohibiting out-of-thin-air behaviors on the Linux kernel. They inject identifiable assembly sequences into the compiler output and use binary rewriting techniques to test different instruction sequences that may prevent out-of-thin-air behaviors. Our work focuses on a more general-purpose approach which involves modifying existing compiler code generation process and comparing the result with the original compiler.

7.5 Other Related Work on Weak Memory Models

The RMC approach Crary and Sullivan (2015); Sullivan (2017) has proposed an approach called *Relaxed Memory Calculus* (RMC) that is fundamentally different from the

C/C++ and Java memory models. In the RMC approach, programmers essentially reason about the relative ordering of memory accesses in concurrent programs (in a fashion close to hardware memory models) and explicitly specify the constraints on the execution order and visibility of writes. Unfortunately, the RMC approach also suffers from OOTA behavior and needs further fixes. It is important to note that Sullivan (2017) demonstrates that ARMv8 seems to have a smaller overhead on dependencies/fences than ARMv7 and Power; more notably, SC atomics perform nearly as well as C11 atomics on ARMv8. Hence, this encourages future work to extend our evaluation to the ARMv7 and Power architectures.

Proposals for other languages There are also relaxed memory model proposals for other high-level programming languages such as Go (goM, 2014), Javascript (jav, 2016), Rust (rus, 2018), Swift (swi, 2017), and WebAssembly (web, 2017). These memory model proposals generally use the C/C++ or Java memory models as their guideline and would likely face the OOTA problem. The two approaches that we study in this dissertation could also serve as potential solutions for these proposals.

Concurrency Semantics for LLVM IR Chakraborty and Vafeiadis (2017) propose a model based on event structures for LLVM intermediate representation, which is stronger than the C/C++ memory model, weaker than known hardware memory models and validates compiler optimizations.

Hardware Memory Models There exist significant research efforts in investigating and formalizing hardware memory models (Sewell et al., 2010; Sarkar et al., 2011; Maranget et al., 2012; Flur et al., 2016; Pulte et al., 2018). While hardware memory models may be subject to changes for future processor design and optimizations, they generally do not allow out-of-thin-air behaviors since they respect a syntactic data and control dependencies, while traditional compiler optimizations could potentially introduce such behaviors (Boehm and

Demsky, 2014). Our dependency-preserving approach defines a dependency notion that is close to that of the hardware and enforces the compiler to generate code that respects such dependencies.

Instantaneous instruction execution framework Zhang et al. (2017) provide a framework based on operational semantics to specify weak (hardware) memory models. Their model introduces hardware abstractions based on monolithic memory, invalidation buffers, timestamps and dynamic store buffers, etc., to capture micro-architectural optimizations in modern processors, and can express SC and TSO.

Denotational Weak Memory Models Castellan (2016) has proposed a denotational semantics based on event structures for a toy concurrent programming language. In his model, a thread is represented by a deordered event structure, and relaxed behaviors are covered with two parts, the processor part and the memory part. The processor part explains the reordering of memory operations, and the memory part explains how memory operations are propagated to the other threads.

Chapter 8

Conclusion and Future Work

Restricting compiler optimizations is a promising solution to eliminate out-of-thin-air behaviors. Our results show that on an ARMv8 processor the dependency-preserving approach has an average overhead of 3.1% and a maximum overhead of 17.6% on the SPEC CPU2006 C/C++ benchmarks, and that the load-store-order-preserving approach has no overhead on average and a maximum overhead of 6.3% on 43 concurrent data structures, which indicates that the approach deserves further consideration. There remain opportunities to further reduce overheads by implementing more sophisticated optimizations and by carefully auditing the compiler optimization passes we omitted.

The two approaches we have studied in this dissertation may inspire future research on investigating the runtime overhead of forbidding OOTA behavior, and some potential directions are listed as follows:

- **Evaluating the overhead on other relevant processors**. The current evaluation is targeted at an ARMv8 processor. As Sullivan (2017) points out, the ARMv8 processors may have a smaller overhead on dependencies/fences than the ARMv7 and Power processors, and hence it would make our study of the two approaches more thorough

to evaluate them on ARMv7 and Power processors.

- **Extending the benchmarks**. We evaluate the dependency-preserving approach on single-threaded and two-copy SPEC CPU2006 benchmarks. One potential improvement is also to evaluate this approach on multi-threaded application benchmarks. We evaluate the load-store-order-preserving approach on relatively small-scale concurrent data structure benchmarks. It may enhance the evaluation to add (single-threaded and multi-threaded) full application benchmarks and high-performance computing benchmarks that use C/C++ relaxed atomics.

- **Implementing and evaluating the dependency-preserving approach for Java**. The dependency-preserving approach is currently implemented and evaluated for C/C++. Given that this approach may potentially fit in the Java context, one potential direction to strengthen the study would be implementing the dependency-preserving approach for Java and evaluating its runtime overhead.

Bibliography

2014. The Go Memory Model. https://golang.org/ref/mem. (May 2014).

2016. ECMAScript Sharedmem: Formal Memory Model Proposal Tracking. https://github.com/tc39/ecmascript_sharedmem/issues/133. (July 2016).

2017. https://github.com/apple/swift/blob/master/docs/proposals/Concurrency.rst. (Jan 2017).

2017. https://github.com/tc39/ecmascript_sharedmem/issues/133. (May 2017).

2018. https://doc.rust-lang.org/beta/nomicon/atomics.html. (2018).

Sarita V. Adve. 2004. The SC- Memory Model for Java. http://rsim.cs.illinois.edu/~sadve/jmm/sc-.pdf. (2004).

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*.

Jade Alglave, Luc Maranget, Paul E McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems*.

ARM. 2016. ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). (2016).

Azul. 2017. https://www.azul.com/press_release/falcon-jit-compiler/. (May 2017).

Mark Batty. 2014. *The C11 and C++11 Concurrency Model*. Ph.D. Dissertation. University of Cambridge.

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015a. The Problem of Programming Language Concurrency Semantics. In *Proceedings of the 2015 European Symposium on Programming*.

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015b. The Problem of Programming Language Concurrency Semantics. In *Proceedings of the 24th European Symposium on Programming*.

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the Symposium on Principles of Programming Languages*.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Pete Becker. 2011. ISO/IEC 14882:2011, Information Technology – Programming Languages – C++. (2011).

Hans Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

Hans-J Boehm. 2005. Threads Cannot Be Implemented as A Library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Hans J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Matko Botinčan, Paola Glavan, and Davor Runje. 2010. Verification of Causality Requirements in Java Memory Model is Undecidable. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics*.

Simon Castellan. 2016. Weak Memory Models Using Event Structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*.

John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the 5th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Proceedings of the 2007 European Symposium on Programming*.

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*.

Karl Crary and Michael J Sullivan. 2015. A Calculus for Relaxed Memory. In *Proceedings of the Symposium on Principles of Programming Languages*.

Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. 2013. Plan B: A Buffered Memory Model for Java. In *Proceedings of the Symposium on Principles of Programming Languages*.

Edsger W. Dijkstra. 2002. The Origin of Concurrent Programming. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Cooperating Sequential Processes, 65–138. `http://dl.acm.org/citation.cfm?id=762971.762974`

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 242–255.

Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. (2014).

Facebook. 2018. `https://github.com/facebook/folly`. (Mar 2018).

Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the Symposium on Principles of Programming Languages*.

Kourosh Gharachorloo, Sarita V Adve, Anoop Gupta, John L Hennessy, and Mark D Hill. 1992. Programming for Different Memory Consistency Models. *Journal of parallel and distributed computing* 15, 4 (1992), 399–407.

James Gosling, Bill Joy, and Guy Steele. 1996. The Java Language Specification. (1996).

John L Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

Radha Jagadeesan, Corin Pitcher, and James Riely. 2010. Generative Operational Semantics for Relaxed Memory Models. In *Proceedings of the 2010 European Symposium on Programming*.

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*.

ISO JTC. 2011. ISO/IEC 9899:2011, Information Technology – Programming Languages – C. (2011).

ISO JTC. 2014. ISO/IEC 14882:2014, Information Technology – Programming Languages – C++. (2014).

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Max Khiszinsky. 2017. `https://github.com/khizmax/libcds`. (Dec 2017).

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++ 11. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 618–632.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*.

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A Volatile-by-Default JVM for Server Applications. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 49.

Jan-Willem Maessen, Arvind, and Xiaowei Shen. 2000. Improving the Java Memory Model Using CRF. In *Proceeding of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Jeremy Manson and William Pugh. 2001. Core Semantics of Multithreaded Java. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. ACM, 29–38.

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

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to The ARM and POWER Relaxed Memory Models. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`. (2012).

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. Out-of-Thin-Air Execution is Vacuous. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html`. (Jul 2016).

Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. Pattern-based Synthesis of Synchronization for the C++ Memory Model. In *Formal Methods in Computer-Aided Design*.

Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++ 11 Memory Model. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Proceeding of the 31st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Peizhao Ou and Brian Demsky. 2015. AutoMO: Automatic Inference of Memory Order Parameters for C/C++11. In *Proceeding of the 30th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Peizhao Ou and Brian Demsky. 2017. Checking Concurrent Data Structures Under the C/C++11 Memory Model. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 136:1–136:29 pages.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 391–407.

Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the 4th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the Symposium on Principles of Programming Languages*.

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *Proceedings of the 31st European Conference on Object-Oriented Programming*.

Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *arXiv preprint arXiv:1606.01400* (2016).

Jeff Preshing. 2018. `https://github.com/preshing/junction`. (Feb 2018).

William Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 conference on Java Grande*. ACM, 89–98.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the Symposium on Principles of Programming Languages*.

David P Reed and Rajendra K Kanodia. 1979. Synchronization with Eventcounts and Sequencers. *Commun. ACM* 22, 2 (1979), 115–123.

Erik Rigtorp. 2017a. `https://github.com/rigtorp/SPSCQueue`. (Feb 2017).

Erik Rigtorp. 2017b. `https://github.com/rigtorp/MPMCQueue`. (Aug 2017).

Carl G Ritson and Scott Owens. 2016. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the Symposium on Principles of Programming Languages*.

Vijay Saraswat. 2004. Concurrent Constraint-Based Memory Machines: A Framework for Java Memory Models (summary). In *Annual Asian Computing Science Conference*. Springer, 494–508.

Vijay A Saraswat, Radha Jagadeesan, Maged Michael, and Christoph Von Praun. 2007. A Theory of Memory Models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Jaroslav Ševčík. 2011. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-memory Concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 22.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

Aleksey Shipilëv. 2016a. Java Memory Model Pragmatics. `https://shipilev.net/blog/2014/jmm-pragmatics/`. (Sep 2016).

Aleksey Shipilëv. 2016b. Java Memory Model Pragmatics. `https://shipilev.net/`. (Oct 2016).

Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-end Sequential Consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*.

Michael J Sullivan. 2017. *Low-level Concurrent Programming Using the Relaxed Memory Calculus*. Ph.D. Dissertation. Carnegie Mellon University.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceeding of the 29th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the Symposium on Principles of Programming Languages*.

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Momchil Velikov. 2012. `http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics`. (Oct 2012).

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22th European Conference on Object-Oriented Programming (ECOOP '08)*.

Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. 2017. An Operational Framework for Specifying Memory Models using Instantaneous Instruction Execution. *arXiv preprint arXiv:1705.06158* (2017).

Yang Zhang and Xinyu Feng. 2013. An Operational Approach to Happens-Before Memory Model. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*. IEEE, 121–128.

Yang Zhang and Xinyu Feng. 2016. An Operational Happens-before Memory Model. *Frontiers of Computer Science* 10, 1 (2016), 54–81.

Appendix A

The Select Set of IR-Level Passes Enabled in Our Dependency-Preserving Compiler

Figure A.1 presents the select set of IR-level passes that we enable in our dependency-preserving compiler. Note that compared to the full set of IR-level passes enabled under full optimizations (`-O3`), this is a relatively small set, which means that there remain opportunities to further reduce the overhead by reviewing the disabled passes.

Pass Name	Is the Pass Modified
simplifycfg	Modified
instcombine	
dse	
licm	
slp-vectorizer	
loop-unroll	
gvn	Unmodified
loop-rotate	
mem2reg	
globalopt	
functionattrs	
tailcallelim	
lower-expect	
sroa	
inline	
forceattrs	
inferattrs	
prune-eh	
adce	
rpo-functionattrs	
elim-avail-extern	
float2int	
strip-dead-prototypes	
globaldce	
constmerge	
deadargelim	
argpromotion	
early-cse	
correlated-propagation	
loop-unswitch	
indvars	
loop-idiom	
loop-deletion	
barrier	
alignment-from-assumptions	

Figure A.1: The select set of IR-level transformation passes that we enable in our dependency-preserving compiler. Note that we globally modify the *InstructionSimplify* analysis to preserve data dependencies and to avoid phi nodes merging, which can affect some of the unmodified passes that rely on it, e.g., the *gvn* pass.

Appendix B

Detailed Results for the Load-Store-Order-Preserving Compiler

Now that Figure 6.4 presents a summary of the overheads of several different strategies of preserving load-store ordering, this chapter presents the detailed results for each concurrent data structure that we used to evaluate our load-store-order-preserving compiler.

B.1 Results for Adding Bogus Conditional Branches on Cortex-A72 Cores

B.1.1 Running with a Single Thread

We first present detailed results for single-threaded execution for preserving load-store ordering using bogus conditional branches. These results best capture the actual overhead that our compiler adds to the code to preserve load-store ordering. The plots show percentage slowdown relative to -O3 compilation. Positive numbers mean that the load-store order preserving version is slower than the -O3 version while negative numbers mean that the load-store order preserving version is faster than the -O3 version.

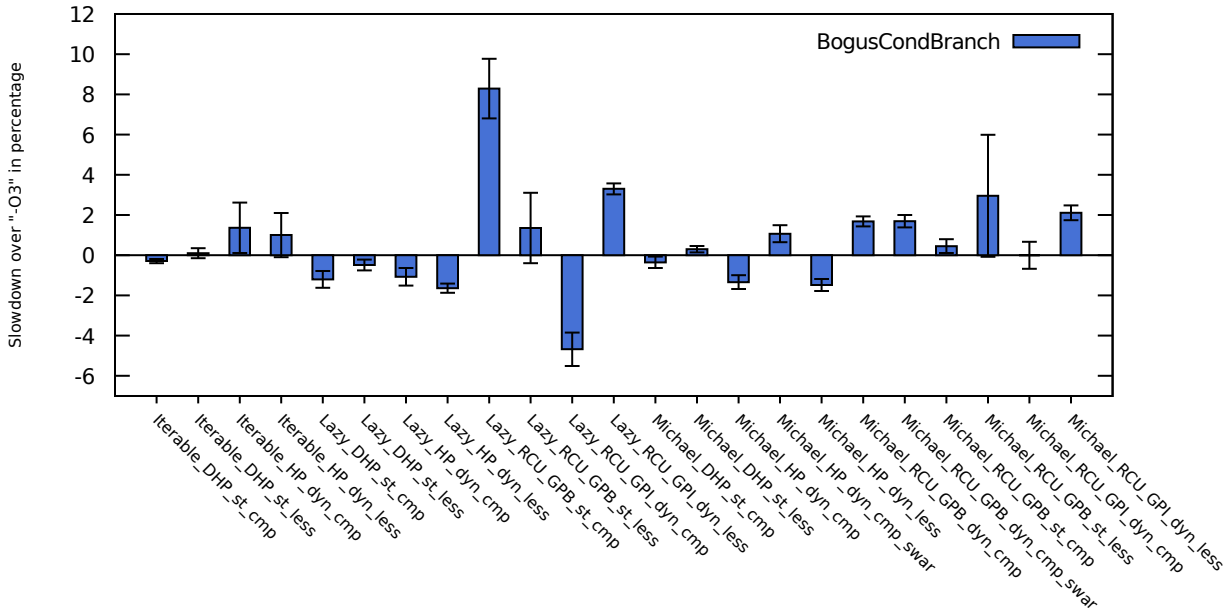

Figure B.1: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different split-ordered list variants from the CDS Library with a single thread.

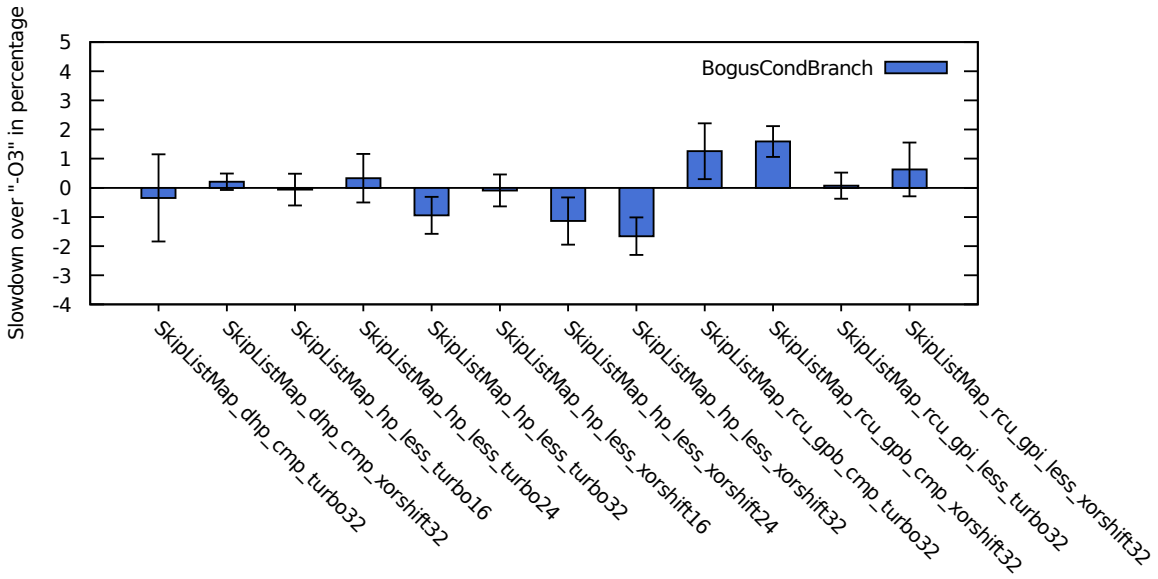

Figure B.2: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different skip list map variants from the CDS Library with a single thread.

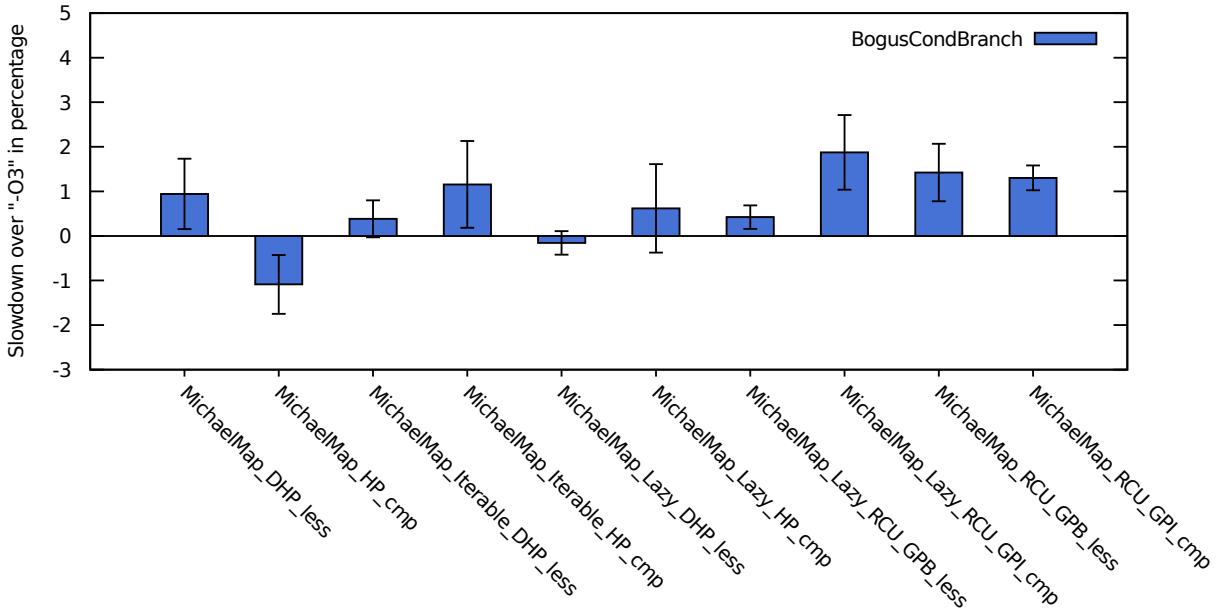

Figure B.3: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different Michael map variants from the CDS Library with a single thread.

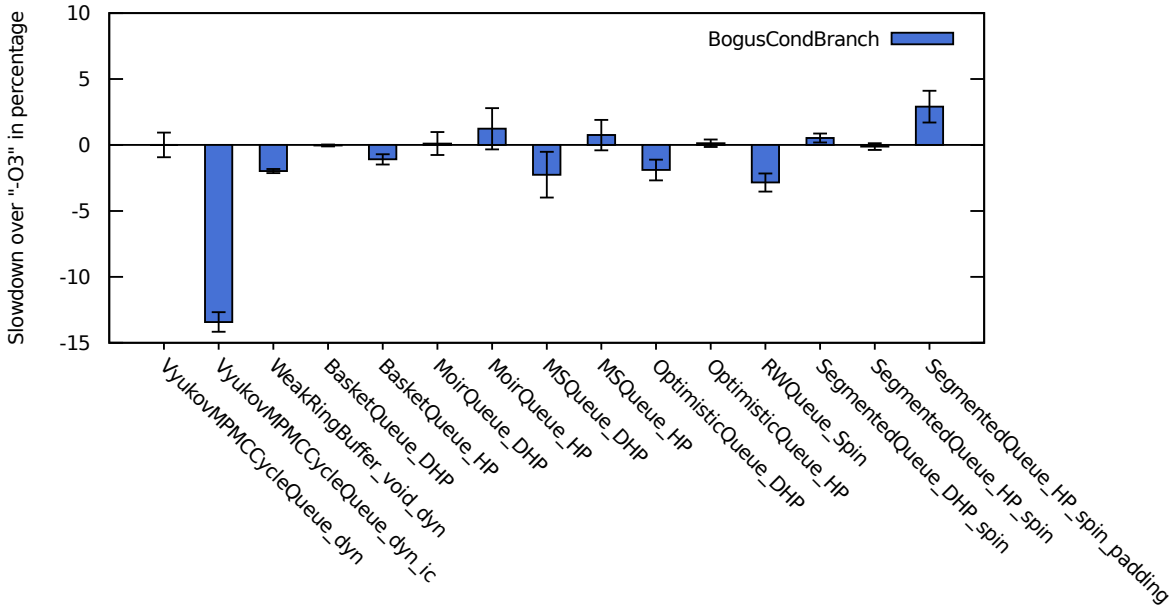

Figure B.4: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different queue benchmarks/variants from the CDS Library with a single thread.

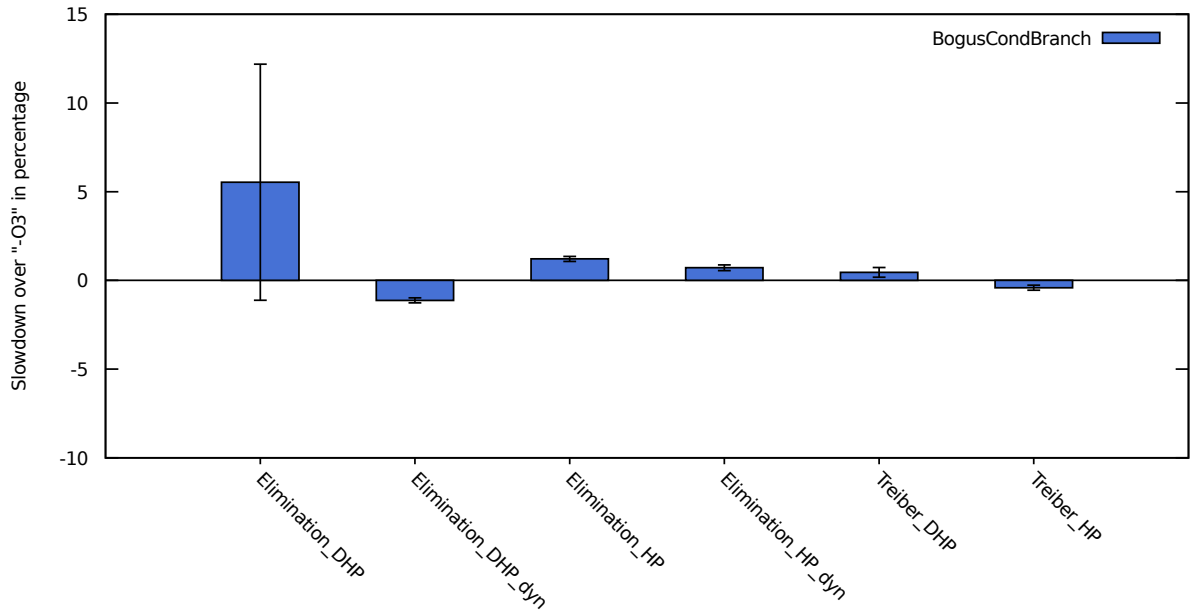

Figure B.5: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for the Treiber stack and elimination-backoff stack variants from the CDS Library with a single thread.

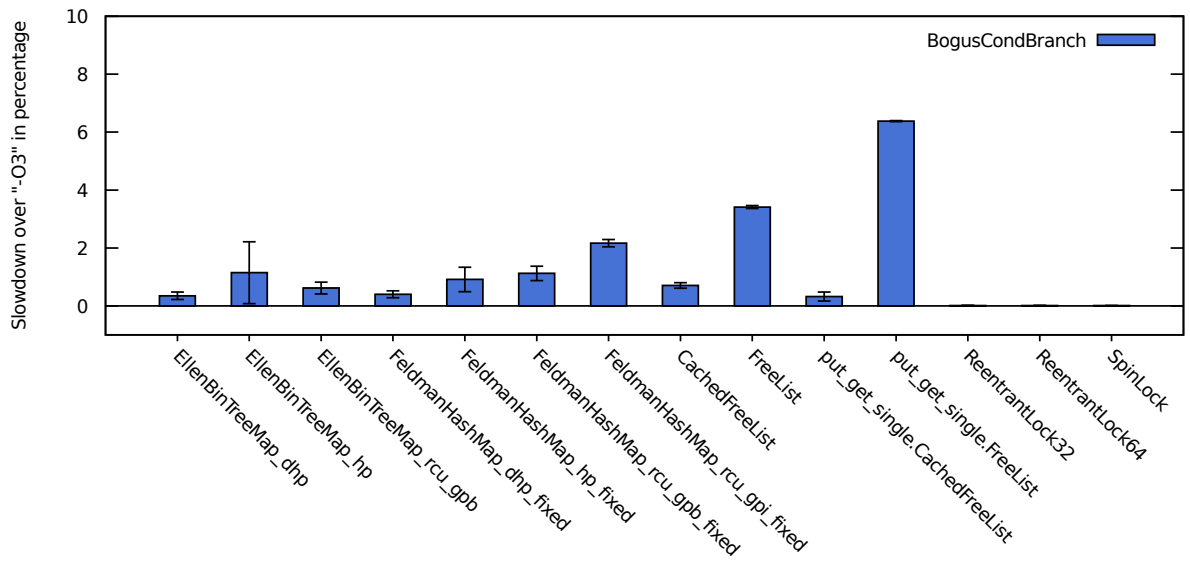

Figure B.6: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for other benchmarks/variants from the CDS Library with a single thread.

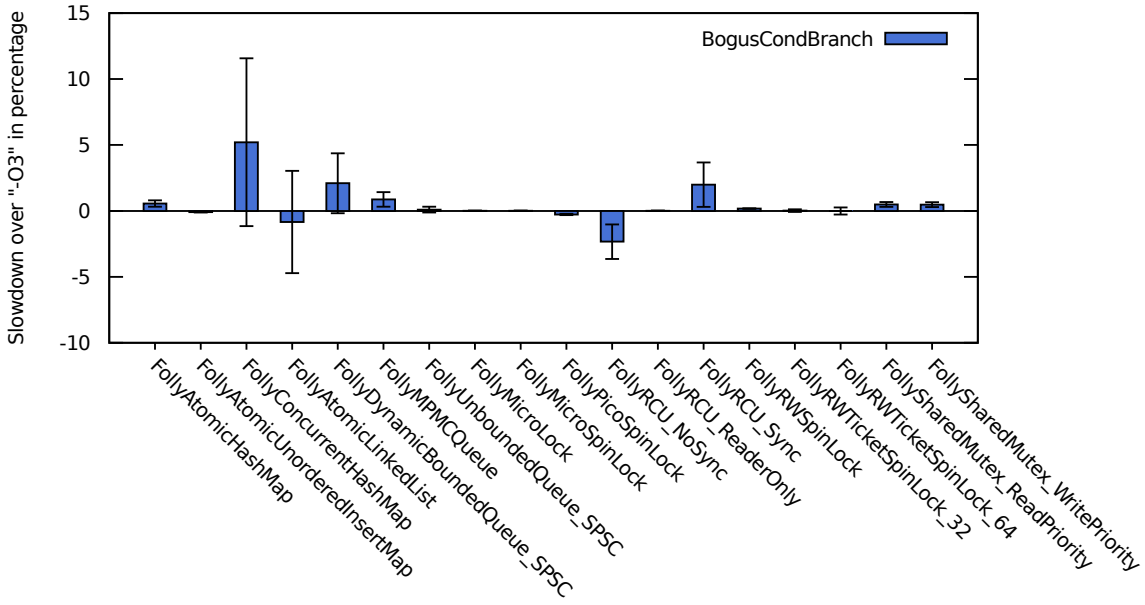

Figure B.7: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks/variants from the Folly Library with a single thread.

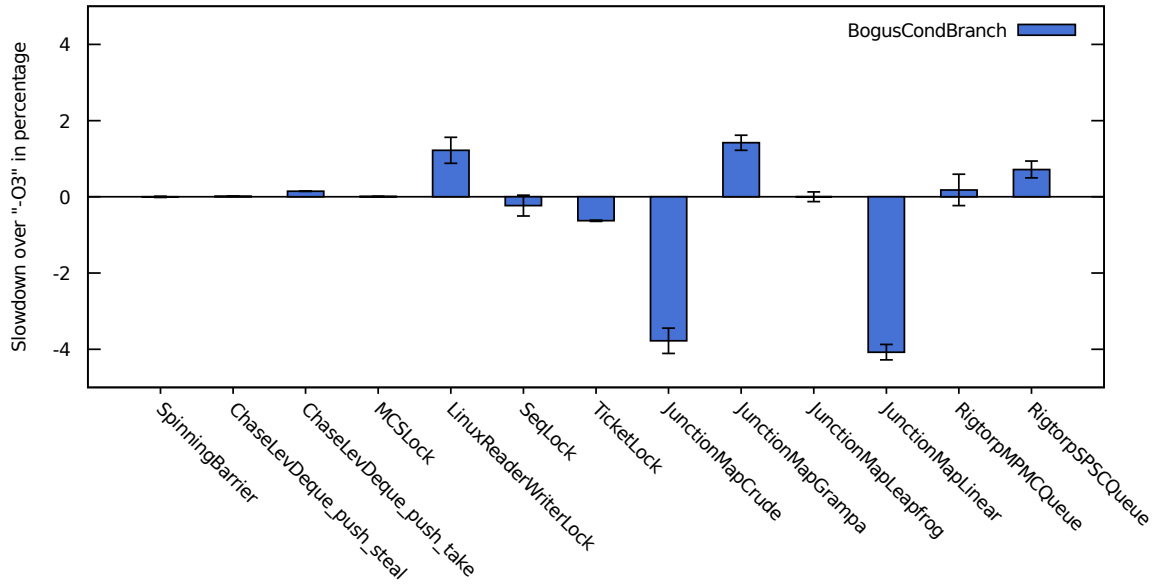

Figure B.8: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks from CDSSpec, Rigtorp's SPSC & MPMC Queues and the Junction Library with a single thread.

B.1.2 Running with Multiple Threads

We next present detailed results for multiple-threaded execution for preserving load-store ordering using bogus conditional branches for completeness. These results are significantly more challenging to interpret as theoretically more efficient code can result in worse performance due to extra contention. The results are also noisy — small differences in timing can result in large performance differences. The plots show percentage slowdown relative to -O3 compilation. Positive numbers mean that the load-store order preserving version is slower than the -O3 version while negative numbers mean that the load-store order preserving version is faster than the -O3 version.

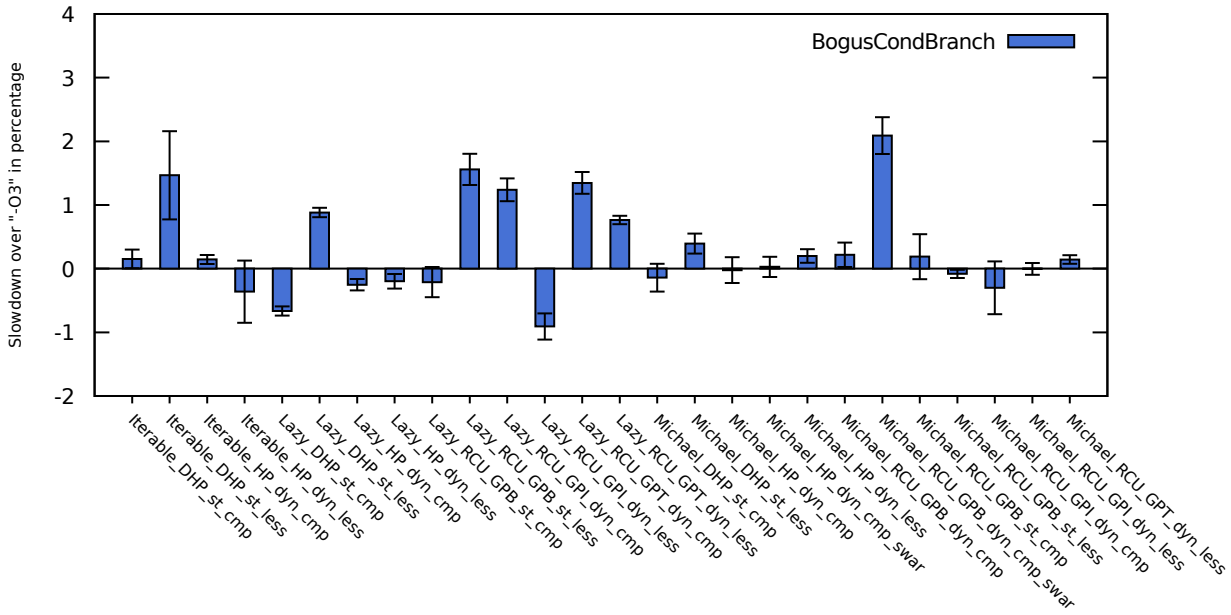

Figure B.9: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different split-ordered list variants from the CDS Library with multiple threads.

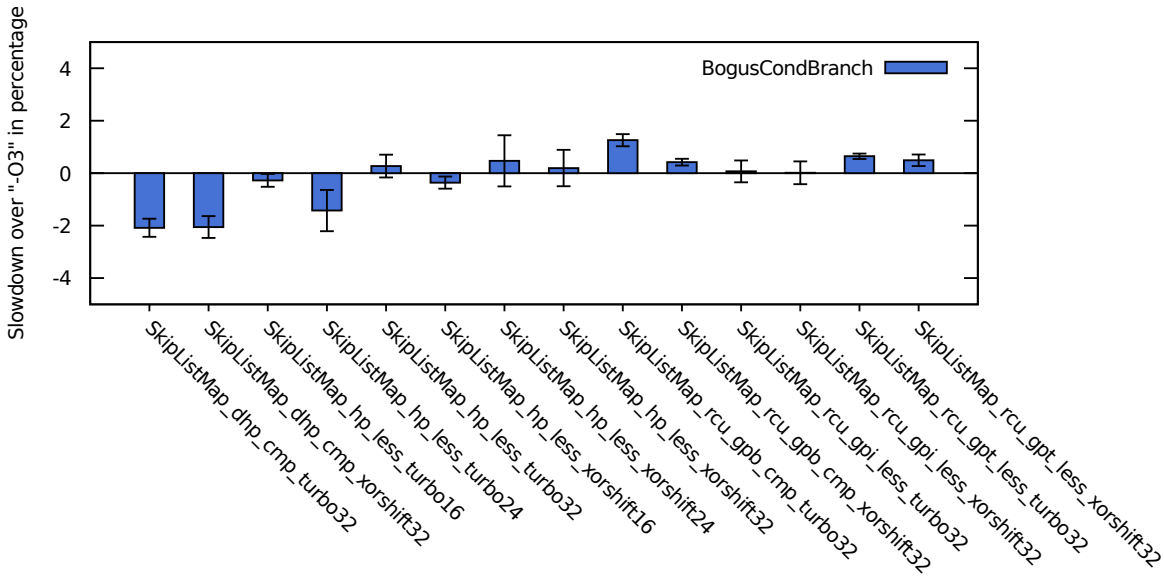

Figure B.10: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different skip list map variants from the CDS Library with multiple threads.

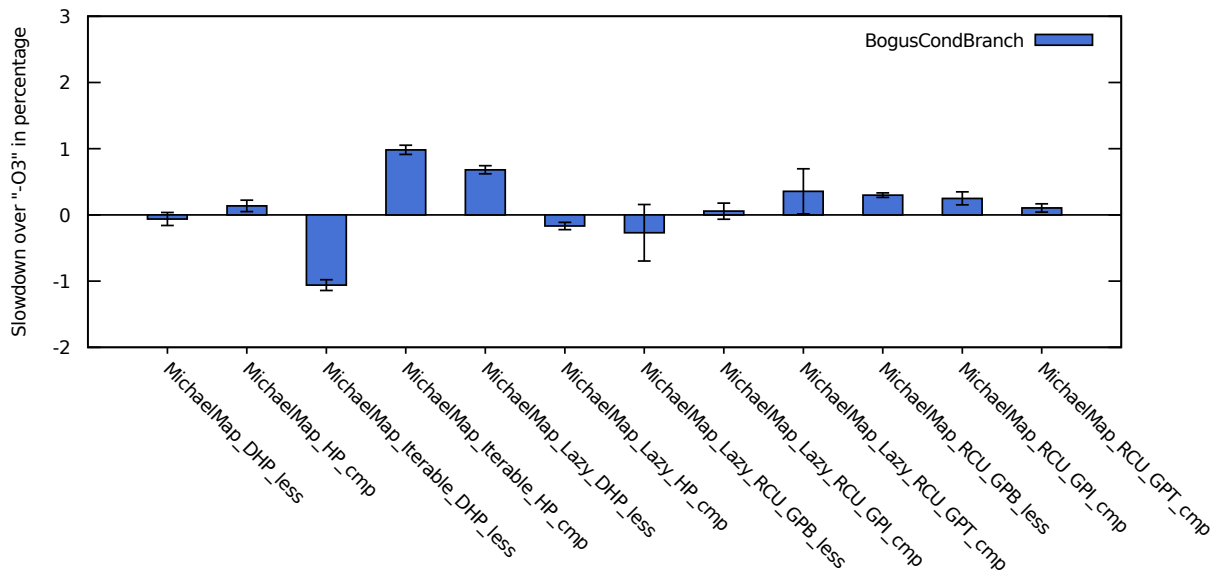

Figure B.11: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different Michael map variants from the CDS Library with multiple threads.

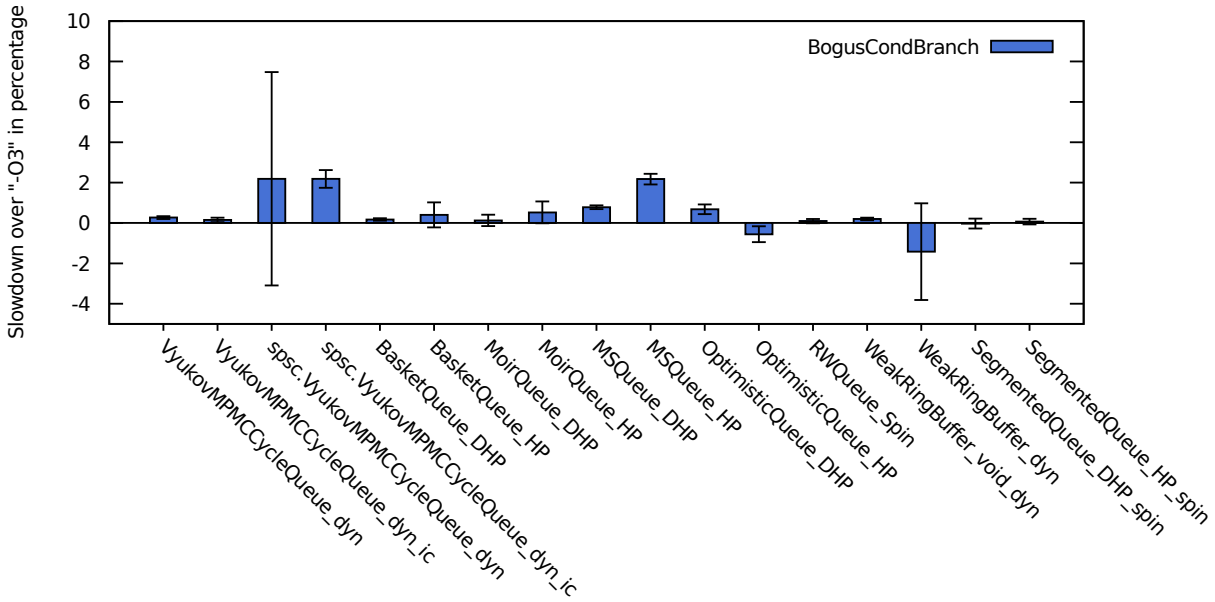

Figure B.12: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different queue benchmarks/variants from the CDS Library with multiple threads.

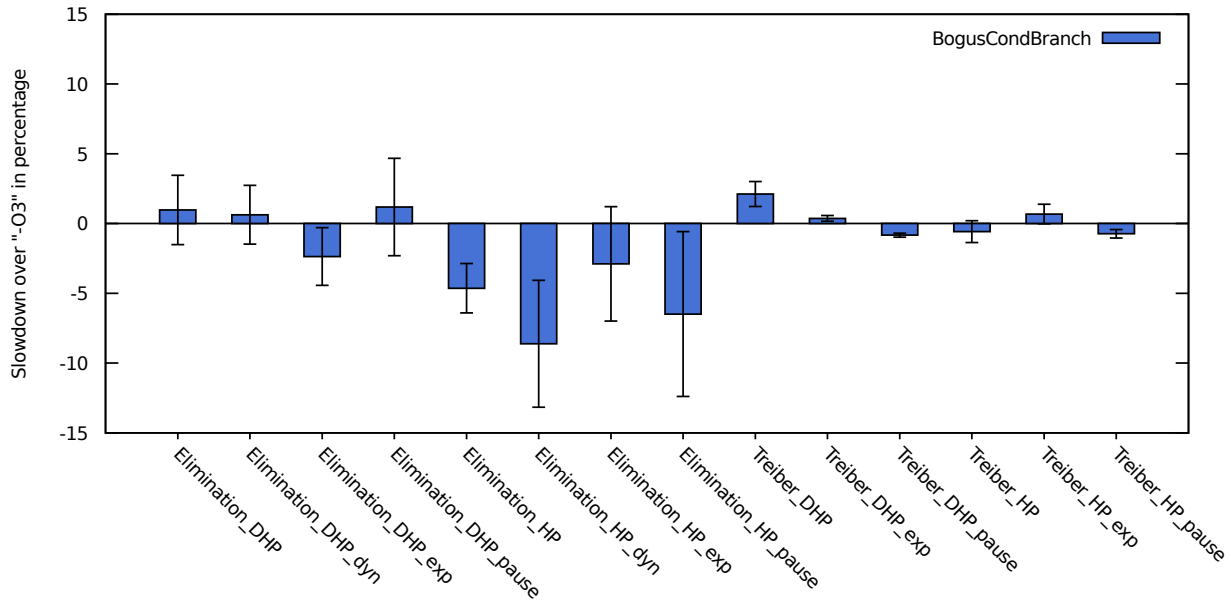

Figure B.13: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for the Treiber stack and elimination-backoff stack variants from the CDS Library with multiple threads.

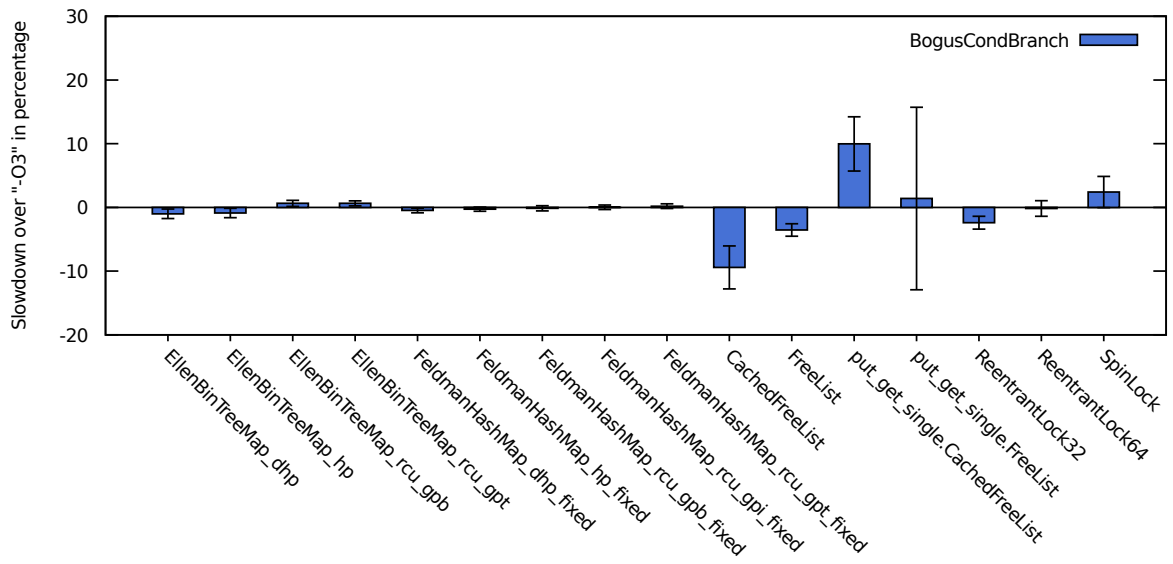

Figure B.14: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for other benchmarks/variants from the CDS Library with multiple threads.

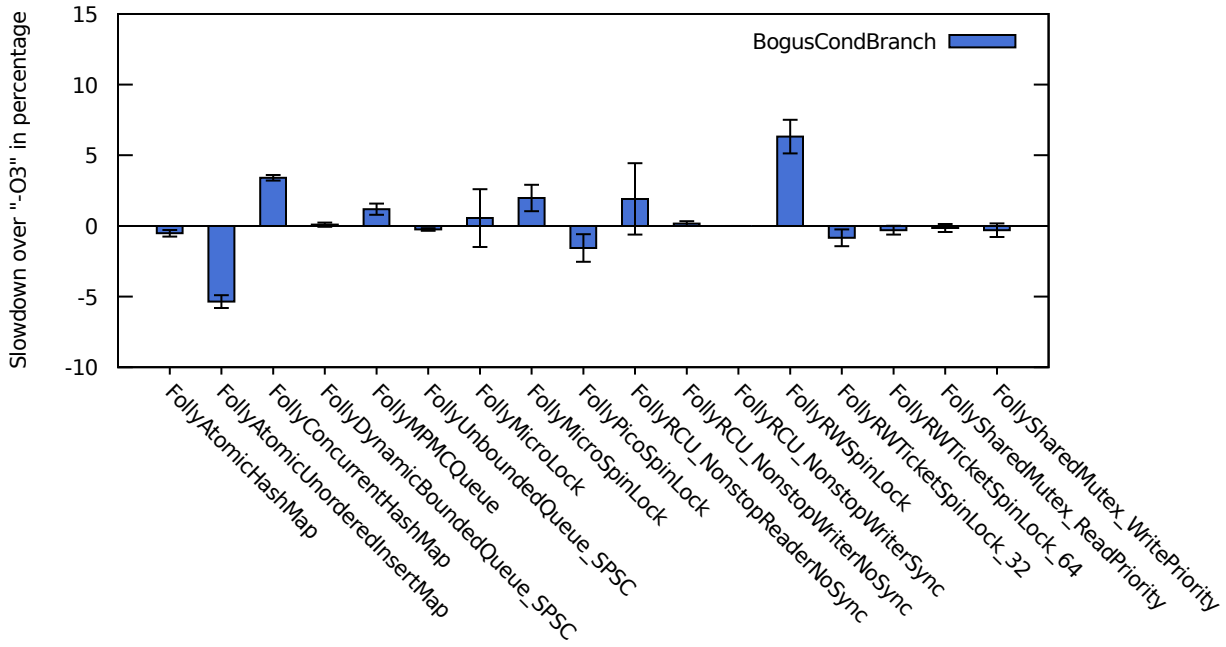

Figure B.15: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks/variants from the Folly Library with multiple threads.

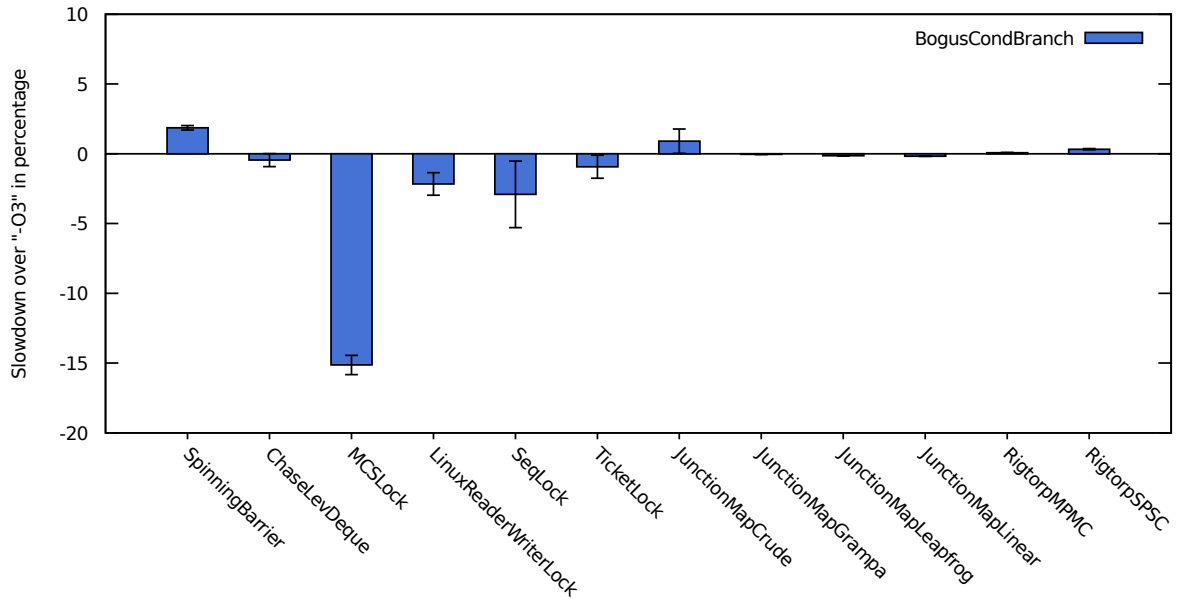

Figure B.16: Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks from CDSSpec, Rigtorp's SPSC & MPMC Queues and the Junction Library with multiple threads.