UNIVERSITY OF CALIFORNIA

Los Angeles

Correctness of Software Tools under Weak Memory Models

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Shuyang Liu

2024

ABSTRACT OF THE DISSERTATION

Correctness of Software Tools under Weak Memory Models

by

Shuyang Liu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Jens Palsberg, Chair

Concurrent programming is often error-prone due to the non-deterministic nature of multi-thread executions. Programmers typically reason about their concurrent programs in the context of *sequential consistency*. Under sequential consistency, instructions in a concurrent program are executed to generate memory events in a linear global order that is consistent with the order specified by the source program, i.e., the program order. However, the assumption of sequential consistency is too strong in practice, given today's processor designs. To achieve better performance, processors today often adopt a *weak memory model*. Under a weak memory model, instructions are not guaranteed to be executed consistently with the program order, due to optimizations such as out-of-order execution and instruction reordering. As a result, previous correctness results of software tools need to be re-visited with considerations of weak memory models. In this thesis, we demonstrate the issues of previous correctness results of software tools and provide new methods for reasoning about their correctness under weak memory models.

The dissertation of Shuyang Liu is approved.

University of California, Los Angeles

2024

To everyone who has spent more than 15 minutes reading through the chapters.

TABLE OF CONTENTS

1 **Introduction** 1

2 **Background** 6

2.1 Execution Graphs 6

2.2 Axiomatic Weak Memory Models 9

2.3 The DRF-SC Guarantee 10

2.4 Compilation, Prediction, and Transformation 12

2.4.1 Compilation 13

2.4.2 Predictive Analysis 16

2.4.3 Transformations 17

3 **Compilation Correctness of Java Access Modes** 22

3.1 The Problem of Compiling Volatile and How to Fix it 24

3.2 Formal Model 29

3.2.1 Basic Syntax 30

3.2.2 The JAM21 Model 31

3.2.3 The JAM21' Model 36

3.3 Compilation Correctness: an Overview 38

3.4 Compilation Correctness to Power 40

3.4.1 The Power Memory Model 40

3.4.2 Compilation Scheme 41

3.4.3 Proof of Compilation Correctness 43

3.5 Compilation Correctness to x86-TSO 44
3.5.1 The x86-TSO Model 44
3.5.2 Compilation Scheme 45
3.5.3 Proof of Compilation Correctness 45
3.6 Performance Implications 46
4 Soundness of Predictive Analyses 49
4.1 Introduction 49
4.2 Motivation 51
4.3 Preliminaries 55
4.3.1 Programs 55
4.3.2 Execution Graphs 56
4.3.3 From Programs to Execution Graphs 58
4.3.4 Bug Sequence 61
4.4 Soundness 62
4.4.1 Composability with bug sequences 64
4.4.2 Well-formedness of Plain Execution 65
4.4.3 Executability 66
4.4.4 Memory Consistency 68
4.5 A Recipe to Prove Soundness 72
4.5.1 Constructing a Symbolic Plain Execution 73
4.5.2 Inserting Consistent Memory Orders 75
4.5.3 Mapping to a Concrete Execution 77
4.6 Proving Race Prediction Algorithms Sound 77

4.6.1 M2 . . . 77

4.6.2 RVPredict . . . 82

4.6.3 MCR-tso . . . 85

4.6.4 Happens-Before (HB) . . . 87

4.6.5 Schedulable Happens-before (SHB) . . . 89

4.6.6 SyncP . . . 90

4.7 Future Work . . . 92

5 Correctness of Transformations under Weak Memory Models . . . 93

5.1 Sound Transformations for Java Access Modes . . . 94

5.1.1 Strengthening . . . 95

5.1.2 Sequentialization . . . 96

5.1.3 Reordering . . . 97

5.1.4 Merging . . . 99

5.1.5 Register Promotion for Non-shared Variable . . . 106

5.1.6 Why are many transformations invalid for Volatile? . . . 109

5.2 Sound Transformations for Predictive Analyses . . . 111

5.2.1 Sound Transformation for x86-TSO . . . 112

5.2.2 Sound Transformation for ARMv8 . . . 113

5.2.3 Using Sound Transformations in Predictive Analyses . . . 115

6 Conclusion . . . 126

A the Full JAM21 Model . . . 127

B the Full JAM21' Model **129**

C The Power Memory Model in Herd7 . . . . **131**

D Proof of Observational Equivalence **134**

E Proof of Compilation Correctness to Power **139**

F The x86 TSO Model in Herd7 . . . . **144**

G Proof of Compilation Correctness to x86-TSO **146**

H Key Properties of the Jam21 Model **150**

H.1 Prior Theorems 150

H.2 Volatile implies SC 153

I The Standard DRF-SC Theorem **157**

J DRF-SC for Execution Graphs **165**

K Experimental Validation of Jam21 . . . . **168**

K.1 Methods supported by Java Architecture for Herd7 168

K.2 Experimental Results 168

K.3 Compilation to Power 174

K.4 Source Code of litmus tests 174

K.5 Full Trace and Litmus Test the example in Section. 3.1 177

L Relationship with Existing Soundness Definitions **181**

M Language Semantics for Chapter 4 . . . . . **185**

N Proofs for Soundness Properties in §4.4 . . . . . **188**

O Proofs of §4.5 . . . . . **196**

P Enhanced-MCR-tso **201**

Q A TSO Race Predictable from SC trace **203**

R TSO Race Discovered by Read Elimination **206**

S WRC-race: another example **209**

T Program Transformations **211**

T.1 Deordering and Reordering 211

T.2 Merging 214

T.2.1 Read-read Merging 214

T.2.2 Write-write Merging 216

T.2.3 Write/RMW-read Merging 218

T.2.4 Write-RMW Merging 223

T.2.5 RMW-RMW Merging 224

T.3 Register Promotion for non-shared Variable 225

References **228**

LIST OF FIGURES

2.1 Store Buffering (SB) 9

2.2 Store Buffering (SB) with Volatile mode accesses 13

2.3 Relationships of Programs and Executions in Compilation, Prediction, and Transformation 14

2.4 Store Buffering (SB) with `MFENCE` inserted 15

2.5 Store Buffering (SB) with a data race 18

2.6 A sequentially consistent execution after transformation witnessing the data race . . 20

3.1 volatile-non-sc.4 under the sequential consistency model, Forbidden 25

3.2 volatile-non-sc.4.ppc translated to Power by HotSpot C1, Allowed 27

3.3 volatile-non-sc.4.ppc translated to Power using the revised compilation scheme, Forbidden 28

3.4 The relationship of programs and executions 39

3.5 Compilation to Power 42

3.6 Compilation to x86-TSO 45

4.1 Various Criteria for a Sound Witness from Past Work 51

4.2 Hierarchy of Existing Sound Definitions under Sequential Consistency 53

4.3 Instructions of the Language 56

4.4 Symbolic Expressions 59

4.5 Predictive Analyses 62

4.6 Similar State Transitions 67

4.7 Race Reporting Criterion of Various Race Prediction Algorithms 78

5.1 The relationship between programs and executions with transformations 94

5.2 Compiler Transformations in C/C++11 and Java 95

5.3 Allowed Deordering Pairs in JAM21 98

5.4 Mergable Pairs in C/C++11 [LVK17] and Java 100

5.5 Before Register Promotion on Volatile access (Forbidden) 109

5.6 After Register Promotion on Volatile access (Allowed) 110

5.7 Example from Fig. 8 in [Pav20] 117

5.8 Transformed SC Execution 119

5.9 A transformation sequence that removes two read events 122

D.1 Cycle Structure caused by Full Fences with Communications 137

K.1 Methods supported by the Java Architecture 169

K.2 volatile-non-sc Experimental Results 170

K.3 Litmus Test Comparisons 171

K.4 IRIW-acq-sc 172

K.5 Z6.U 172

K.6 IRIW-seq-rlx 172

K.7 volatile-non-sc on Power with the incorrect compilation scheme 174

Q.1 Example from Fig. 8 in [Pav20] 204

R.1 A TSO Race Captured by Race Predictor after Read Elimination 207

S.1 A Data Race under the ARMv8 model 210

T.1 Execution Graph before read-read merge on Volatile (Forbidden) 216

T.2 Execution Graph after read-read merge on Volatile (Allowed) 216

T.3 Execution graph before write-write merge on Volatile (Forbidden) 219

T.4 Execution graph after write-write merge on Volatile (Allowed) 219

T.5 Execution Graph before Write-read Merge Transformation (Forbidden) 222

T.6 Execution Graph after Write-read Merge Transformation (Allowed) 222

VITA

2018	B.S. (Computer Science), University of Rochester.
2019–2024	Teaching Assistant, Computer Science Department, UCLA. Compiler Construction course under direction of Professor Jens Palsberg.
2022-2023	Intern, Java Efficiency Team, Meta Inc.
2024	Part-time Faculty Lecturer, Computer Science Department, LMU. Programming Language Semantics course
2018–2024	PhD Student, Computer Science Department, UCLA.
2024–present	Research Scientist, X-Sec Detect and Respond Team, Meta Inc.

CHAPTER 1

Introduction

Concurrent programs have become popular since the multi-thread revolution. Shared-memory concurrent programming, among the concurrent programming paradigms, is a programming style such that threads communicate via a shared memory. Reasoning about shared-memory concurrent programs, however, is notoriously hard due to the non-deterministic nature of multi-thread executions. In a shared-memory concurrent program, instructions from different threads may take effect in many different orders. Programmers without advanced knowledge of the underlying hardware systems typically reason about their concurrent programs relying on an implicit assumption of *sequential consistency* [Lam79]. Under sequential consistency, each memory event generated from the instructions takes effect immediately in the order specified by the source program. In other words, the instructions are executed in a global linear order that is consistent with the order specified in the program.

To see an example, consider the program

initially `x == y == 0`	
Thread 1	Thread 2
`x := 1`	`y := 1`
`r1 := y`	`r2 := x`

In this program, there are two threads, each executing two instructions. The memory locations `x` and `y` are shared between the two threads whereas `r1` and `r2` are local registers. Thread 1 performs a store to a memory location `x` with a value of `1` and then load the value of another memory location `y` to a local register `r1`; Thread 2 performs a store to a memory location `y` with a value of `1` and then load the value of another memory location `x` to a local register `r2`. Under sequential consistency, one can see that at least one of the two registers will receive a value of `1` after the program finishes. Indeed, `r1` receives a value of `0` when loading `y`, then it must be executed before the store of `y` on Thread 2. By that time, the store of `x` is already executed by Thread 1 with a value of `1`. Therefore, at the time when Thread 2 executes the load of `x`, which occurs after the store of `y`, `x` is set to `1` and `r2` would receive `1`. Similarly, we can follow the same reasoning process symmetrically to see that if `r2` receives a `0`, then `r1` must get a value of `1`. As a conclusion, at least one of the two registers has a value of `1` after this program is finished.

However, the assumption of sequential consistency is quite strong in practice. To achieve better performance, hardware and compilers today typically apply many types of optimizations, such as out-of-order execution and instruction reordering. As a result, instructions are not guaranteed to take effect in an order consistent with the program order. In other words, the assumption that there is a global linear order consistent with the program order of which the events are generated by executing instructions no longer holds in modern multiprocessor systems. In the previous example, it is actually possible for both of the registers to receive `0` values after the program finishes its execution on a hardware system with Intel or AMD x86 processors. This can result from the effect of *write buffering* in the x86 architecture. Effectively, each x86 processor has a FIFO buffer of pending stores to the main memory. When Thread 1 executes the load instruction `r1 := y`, its store instruction `x := 1` may still be pending in the write buffer of Thread 1's processor. As a result, when Thread 2 executes the load instruction `r2 := x`, if the store of `x` still has not propagated to the main memory, `r2` would receive a value of `0`. As we have seen, the previous property that "at least one of the registers would get a value of `1`" does not necessarily holds for programs executing on x86 processors. In fact, more counter-intuitive behaviors can be observed on today's hardware systems, such as mobile devices with the ARM chips and server infrastructures with the PowerPC architecture.

In the presence of various optimizations, it becomes critical to specify the instruction semantics to accommodate the execution behaviors that are not included under sequential consistency. Traditionally, these are informally documented in the architecture manuals of the processors [Cor08, Dev07, Cor03]. Since compilers apply optimizations to programs as well, programming language manuals [Lea18, ISO98] also document in English the language semantics in the presence of the potential optimizations.

On the other hand, there have been efforts and success to formalize the program semantics in the presence of possible optimizations [PFD18, ADG21, AFI09, SSO10, SSA11, ML21, BMO12, BP19, MPA05, LVK17] in recent research community. A formal specification for concurrent programs with shared memory, called a *weak memory model*, rigorously describes what behaviors a concurrent program may exhibit in the presence of the possible optimizations.

At the same time, however, many software tools are still implemented by either assuming sequential consistency for simplicity or referring to the informal descriptions from the instruction manuals due to the complexity of the formal weak memory models. Consequently, there is gap between their correctness results and the semantics specified by the formal weak memory models. To close this gap, it is necessary to use the formal weak memory models to show the correctness of software tools. In particular, I focus on three issues in software tooling.

1. **The Correctness of Compilation.** What does it mean to implement a compiler correctly in the presence of a weak memory model? Intuitively, a correct compiler should not introduce any new behavior that is unexpected by the programmers who write the source code. That is, all possible behaviors of the target code after compilation should be expected at the source level. Under sequential consistency, this can be ensured by series of local reasoning on each thread separately. However, under weak memory models, global reasoning is needed since some unexpected program behavior may only be observable when threads are composed together. In addition, programming languages at the source level often adopt different memory models than the architecture at the target level to be multi-platform. Ensuring compilation correctness with respect to the source and target level memory models is not trivial.

2. **The Correctness of Predictive Program Analyses.** A predictive analysis takes an execution trace as input and analyzes the trace without accessing any program source code to predict potential concurrency bugs. Recent work in the area of predictive analysis [MPV21, HMR14, CYW21, Pav20, KP18, TAC23, SWY11, SRA05, SCR13, GRX19] typically use *soundness* as their correctness criteria. A sound predictive analysis yields no false positive, which means every bug that is reported is a real bug of the program. To ensure soundness, existing works have used a wide variety of soundness criterion that characterize correct traces and then prove that the algorithms always produce traces that satisfy the respective soundness criteria. For simple predictive analyses under sequential consistency, this tends to be sufficient to convince a reader about soundness. However, these trace-based approaches fit poorly under weak memory models and have implicit assumptions on the program semantics.

3. **The Correctness of Program Transformations.** Lastly, program transformations are commonly seen in compilers. Compilers transforms program to achieve better performance of the target code. In addition, transformations can be applied to explain some weak memory semantics in sequential consistency [LV16]. Which transformations will maintain the correctness of compilers and predictive analyses? While there has been extensive research on correct transformations under the C/C++11 memory model [LVK17, VBC15], the original Java Memory Model (JMM) [Še08, MPA05], and common hardware models [LV16], none focuses on the memory model of the Java Access Modes (JAM), which is introduced since Java 9. Despite the similarities between JAM and the C/C++11 atomic operations, there are subtle differences in their semantics with respect to transformations. But these subtle differences are only informally documented [Lea11a, Lea18]. Moreover, to the best of our knowledge, transformations in predictive analyses have not been not formally studied in existing works.

All three types of software tools described above can be seen as functions of executions, which are abstract structures that capture the run-time semantics of concurrent programs. More precisely, a compiler takes an input program written in a source language A and produce a new program in the target language B. From the perspective of executions, it can be seen as a function that maps a set of executions of the input program that are consistent under the memory model of A to a set of executions with the same observable behaviors that are consistent under the memory model of B. In the case of transformations, each transformation can be seen as a function that maps a set of executions consistent under the memory model of a language A to another set of executions with the same behaviors that are consistent under the same language A. Predictive analyses, on the other hand, focus on a single execution each time. A predictive analysis takes a single execution of the program that is consistent under sequential consistency and predicts a bug-revealing execution of the same program that is consistent under the memory model of some language A (which may be sequential consistency as well). Therefore, we can study their correctness properties under weak memory models at the level of *execution graphs*, which is a data structure used in the weak memory research field to formally represent executions.

Overall, the thesis statement is the following.

> *It is possible to formally show the correctness of software tools under weak memory models.*

The rest of this thesis is organized in the following way. In Chapter 2 we introduce the key concepts that are essential in this thesis; in Chapter 3, we present a bug in the Java compiler and show how to fix the bug to ensure compilation correctness under memory model of the Java Access Modes; in Chapter 4, we propose a new soundness definition that can be used under weak memory models and provide a recipe to construct a proof for such a soundness theorem; in Chapter 5, we prove a set of program transformations correct for the Java compiler with respect to the memory model of the Java Access Modes and a show a set of provably sound transformations one can use to predict concurrent bugs under weak memory models. Lastly, Chapter 6 concludes the thesis.

CHAPTER 2

Background

In this chapter, we introduce some of the key concepts that are commonly used in concurrency literature. While we defer the formal definitions to later chapters, the goal of this chapter is to introduce the terminologies and provide a foundation for later chapters.

2.1 Execution Graphs

An **execution graph** is a data structure that describes a concurrent execution with sets and relations. Each execution graph consists of the following basic components:

- An **event set** (Evts). An event can be seen as an effect of executing an instruction. Depending on the specific semantics of the instruction, different types of event can be generated. Typically, in the context of concurrency, each store instruction generates a single write event and each load instruction generates a single read event. Instruction sets of modern architectures such as x86 or ARM also include synchronization mechanisms such as fence instructions and atomic instructions. These instructions typically generate their own types of events or read or write events with special attributes. For programming languages, it is possible for a statement or an expression to generate multiple events in a specified evaluation order.
- **Program order** (po). Events generated on the same thread are linearly ordered by the program order. Program order on each thread represents the local control flow unfolded by executing each instruction in the order specified by the source program.

- **Reads-from order** (`rf`). For each read event in the event set, its value is determined by a write event accessing the same memory location. Note that this write event can be originated from a thread other than the thread of the read event. The `rf` order relates such a write to a read event. While there may be multiple read events relate to the same write event by `rf`, each read event is related to a unique write event. In addition, the semantics of `rf` requires that each pair of write and read events related by `rf` access the same memory location with the same value.

- **Coherence order** (`co`). For each memory location, due to cache coherence protocols, there is a global consensus on the order of writes to this location. The `co` order is a union of total orders that relate each pair of write events accessing the same location.

- **From-reads order** (`fr`). If a read event reads from a write event that is `co`-ordered before another write, then the read must have been executed before the latter write. The `fr` order captures this notion by relating this read to the latter write.

- **Dependencies**. Modern architectures such as x86 [SSO10], ARM [AFI09, ADG21], and PowerPC [AFI09, SSA11] respect syntactic dependencies imposed by the instructions. If the memory location that an instruction accesses is determined by the result of executing a previous load instruction on the same thread, then there is an address dependency between the events generated by the two instructions; if the value of a store instruction is determined by the result of executing a previous load instruction, then there is a data dependency between the events generated by the two instructions; lastly, if the control flow of which an event is generated on is determined by the result of executing a previous load instruction, then there is control dependency between the read event generated by the load instruction and the events generated on this control flow after the read. Note that syntactic dependencies may not be respected by compilers due to optimizations such as *common sub-expression elimination*. However, distinguishing syntactic dependencies and semantic dependencies requires complex global reasoning about the program, which is hard to specify in a programming language model. This issue leads to the famous *out-of-thin-air* problem, which remains a challenge in the research field.

In addition, there are additional types relations among events specific to the instruction set architecture and the programming language. In Chapter 3, we will formally introduce above relations and the additional relations defined for Java, x86, and PowerPC.

Each program is mapped to multiple execution graphs due to the non-determinism of concurrency. We will explain the formal relationship between a program and the set of its execution graphs in Chapter 4.

Example To see an example, consider the execution graph in Fig. 2.1, which demonstrates an execution with the write buffering effect described in the previous chapter and is often called *Store Buffering* in many weak memory literature. In this execution graph, we use $\mathtt{Wx} = n$ to denote a write event storing to a memory location x with a value of $n \in \mathbb{N}$ and $\mathtt{Rx} = n$ to denote a read event reading from a memory location x that receives a value of $n \in \mathbb{N}$. The event set of this execution graph is $\{\mathtt{Wx} = 0, \mathtt{Wx} = 1, \mathtt{Wy} = 0, \mathtt{Wy} = 1, \mathtt{Rx} = 0, \mathtt{Ry} = 0\}$. Events stem from the same thread are ordered by $\mathtt{po}$. In this execution graph, $\mathtt{po} = \{\langle \mathtt{Wx} = 1, \mathtt{Ry} = 0\rangle, \langle \mathtt{Wy} = 1, \mathtt{Rx} = 0\rangle\}$. Each memory location is initialized to 0 by the initial write events $\mathtt{Wx} = 0$ and $\mathtt{Wy} = 0$, which are $\mathtt{co}$-ordered before the other write event of the same memory location. Since both read events get a value of 0, they read from the initial write events, respectively. Hence we have $\mathtt{Wx} = 0 \xrightarrow{\mathtt{rf}} \mathtt{Rx} = 0$ and $\mathtt{Wy} = 0 \xrightarrow{\mathtt{rf}} \mathtt{Ry} = 0$. Since other write events are $\mathtt{co}$-ordered after the initial write events, the read events are $\mathtt{fr}$-ordered before these write events. Hence, we have $\mathtt{Rx} = 0 \xrightarrow{\mathtt{fr}} \mathtt{Wx} = 1$ and $\mathtt{Ry} = 0 \xrightarrow{\mathtt{fr}} \mathtt{Wy} = 1$. Note that there is a cycle formed by $\mathtt{fr}$ and $\mathtt{po}$ orders in the execution graph (highlighted with thick edges in Fig. 2.1): $\mathtt{Wx} = 1 \xrightarrow{\mathtt{po}} \mathtt{Ry} = 0 \xrightarrow{\mathtt{fr}} \mathtt{Wy} = 1 \xrightarrow{\mathtt{po}} \mathtt{Rx} = 0 \xrightarrow{\mathtt{fr}} \mathtt{Wx} = 1$. Lastly, there is no dependency present in this execution graph.

$$\begin{tikzcd}
 & \mathtt{Wx} = 0 \arrow[dl, "\mathtt{co}"'] \arrow[ddrr, "\mathtt{rf}" near end] & \mathtt{Wy} = 0 \arrow[dr, "\mathtt{co}"] \arrow[ddll, "\mathtt{rf}"' near end] & \\
\mathtt{Wx} = 1 \arrow[d, "\mathtt{po}"'] & & & \mathtt{Wy} = 1 \arrow[d, "\mathtt{po}"] \\
\mathtt{Ry} = 0 \arrow[urrr, "\mathtt{fr}"] & & & \mathtt{Rx} = 0 \arrow[ulll]
\end{tikzcd}$$

Figure 2.1: Store Buffering (SB)

2.2 Axiomatic Weak Memory Models

Traditionally, program semantics under weak memory are defined in an operational style. For example, to formally describe the write buffering effect that we have seen previously on x86 architecture, the operational semantics for the x86 architecture [SSO10] defines a state machine consists of multiple processors with a write buffer on each processor and a main shared memory. To execute an instruction, the state machine has to go through multiple transitions from fetching the instruction to complete the instruction. Similarly, the *Flat model* [PFD18] is an operational model that describes the program behaviors on ARMv8 machines with a state machine and a set of transitions.

However, these operational models often lead to complex reasoning process when determining which program behaviors are observable. Moreover, it is hard to compare different operational models for weak memory systems in terms of the program behaviors that they allow. In recent years, using **axiomatic** [AMS12] models has become a more popular approach to formally specify the program behaviors in weak memory systems. An axiomatic memory model is a set of assertions over complete execution graphs. An execution graph is called *consistent*, if all assertions specified by the axiomatic memory model hold. In this case, it means the execution presented in the graph may be observable under the memory model. Otherwise, the execution is said to be *inconsistent*, which means it is guaranteed to be unobservable under the memory model. Comparing to the operational memory models, axiomatic memory models allow simpler reasoning process to determine whether certain behaviors is observable and they are recently formalized in the language of Kleen Algebra with Test (KAT) [KLV23].

Example Among all the axiomatic memory models, the simplest model is **sequential consistency (SC)**. Under sequential consistency, each event takes effect atomically and there is a total order in which the events occur. Hence, orders shown in the execution graph should be acyclic. In Fig 2.1, we see that there is a cycle formed the `fr` and po orders. Therefore, the execution shown in Fig 2.1 is said to be forbidden under SC. In other words, such an execution is never observable on machines that guarantee SC. Note that this is consistent with the explanation from the previous chapter: under SC, at least one of the read events gets a value of 1. On the other hand, weak memory models such as the **x86-TSO** [OSS09] model allows such cycle to occur in an execution graph. This is achieved by excluding the po orders between a write event and a read event accessing different memory location from the set of orders that need to be acyclic. In Fig. 2.1, both of the po orders are excluded and there is no cycle formed after excluding them. Therefore, the execution shown in Fig.2.1 is allowed under x86-TSO. In other words, it may be observed on x86-TSO machines.

2.3 The DRF-SC Guarantee

Despite all the counter-intuitive program behavior that a weak memory model allows, all architectures adopting weak memory models should provide sufficient synchronization mechanisms for programmers to *restore* sequential consistency. Modern architectures achieve this by providing synchronization instructions that enforce orders among events. On x86 architecture, fence instructions, such as the `MFENCE` instruction, can be used to make sure the instructions before a fence are completed before the instructions after the fence start. Effectively, the po orders between the two groups of instructions are preserved. Similarly, the ARMv8 instruction set provides

DMB instructions for enforcing the orders among instructions. In addition to the fence instructions, modern architectures also provide other synchronization primitives that guarantee atomic access to memory locations. In the x86 instruction set, the LOCK ADD instruction can be used to atomically update a memory location. In the ARMv8 instruction set, the LDXR and STXR instructions can be used to build atomic access when paired together. In execution graph, these primitives usually correspond to read-modify-write (RMW) events or pairs of read-write events that carry out special synchronization semantics. Effectively, there is a linear order enforced among the atomic accesses of each memory location. Therefore, they can also be used to restore sequential consistency. The semantics of these synchronization primitives are typically formally defined in axiomatic memory models as well.

In order to show that a weak memory model provide sufficient mechanisms to restore sequential consistency, a theorem that serves as a correctness criteria for weak memory models, called the **DRF-SC theorem** [AH90], has to be proven. The DRF-SC theorem gives a guarantee that follows programmers' basic intuition: *if a program is properly synchronized, then all its executions are sequentially consistent.* By "properly synchronized", the DRF-SC theorem requires that there is no *data race* present in the program. A data race occurs when two threads access the same memory location concurrently and one of them is writing to the location. It indicates that there is no order enforced between the two accesses and is often a root cause of concurrency bugs. Based on different focuses and various definitions of data race, different versions of the DRF-SC theorem are seen in existing literature [BMO12, LVK17, WPP20, BP19]. As we will see in Chapter 3, we first prove a version of the DRF-SC theorem, which specifies that data-race-freedom in *all* sequentially consistent executions of a program implies the program does not have other weak memory executions. A slightly weaker version originally seen in [BP19] is also proved in Chapter 3, which states that if an execution graph imposes no happens-before race, then it is sequentially consistent. In Chapter 4, we show another version of the theorem stating that enclosing memory accesses in critical sections protected by locks can also restore sequential consistency.

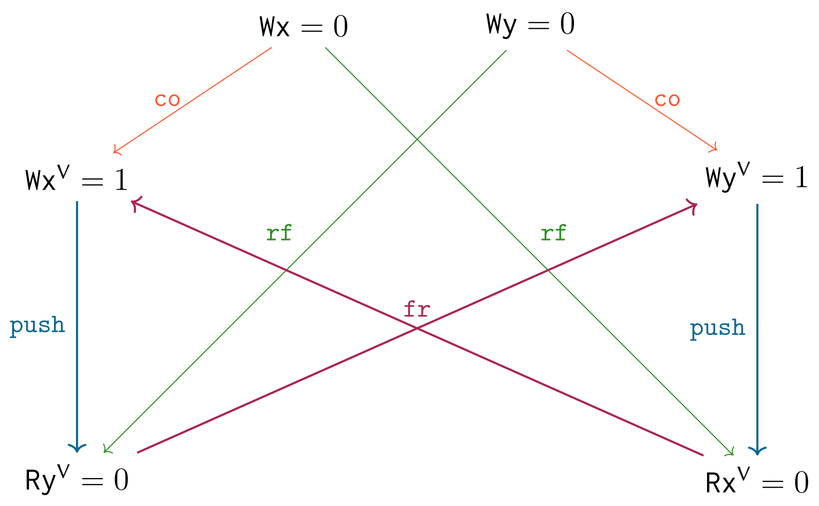
Example In the example of Fig 2.1, there are two pairs of data races: $\langle \mathtt{Wx} = 1, \mathtt{Rx} = 0\rangle$ and $\langle \mathtt{Wy} = 1, \mathtt{Ry} = 0\rangle$. Due to the presence of these two data races, the weak memory behavior shown in Fig. 2.1 is allowed under the x86-TSO model. In fact, it is also allowed under the memory model of the Java Access Modes. The Java Access Modes provide Volatile mode operations, which can be used to eliminate data races. If we mark all of the accesses in the previous program as Volatile accesses as the following:

initially `x == y == 0`	
Thread 1	Thread 2
`x.setVolatile(1);`	`y.setVolatile(1);`
`r1 = y.getVolatile();`	`r2 = x.getVolatile();`

Then the two data races are eliminated from the program. The Volatile operations in the Java Access Modes brings synchronization effects that enforces certain program orders, which result in the execution graph shown in Fig. 2.2. In this execution graph, the program orders enforced by the Volatile write events $\mathtt{Wx}^{\mathsf{V}} = 1$ and $\mathtt{Wy}^{\mathsf{V}} = 1$ are represented as the `push` order. The previous cycle formed by po and `fr` orders is now a cycle formed by `push` and `fr`. The model of the Java Access Modes specifies that such a cycle is *forbidden*. Thus, this execution is prevented under the model of the Java Access Modes after we mark the accesses with Volatile. In fact, now all the executions of this program are sequentially consistent. That is, the set of executions of this program under the model of the Java Access Modes now coincides with the set of executions under sequential consistency. In general, Volatile mode is used to restore sequential consistency in Java.

2.4 Compilation, Prediction, and Transformation

While it seems that the three aspects we focus on, compilation, prediction, and transformation, are each itself a separate topic, we explain how their correctness can be studied using the same method with execution graphs in this section.



$\mathsf{Wx} = 0$
$\mathsf{Wy} = 0$
co
co
$\mathsf{Wx}^{\mathsf{V}} = 1$
$\mathsf{Wy}^{\mathsf{V}} = 1$
rf
rf
fr
push
push
$\mathsf{Ry}^{\mathsf{V}} = 0$
$\mathsf{Rx}^{\mathsf{V}} = 0$

Figure 2.2: Store Buffering (SB) with Volatile mode accesses

2.4.1 Compilation

Compilation is an operation of translating a source program to a target program by applying a compilation scheme, which is a function that maps source-level instructions to target-level instructions. As shown in Fig. 2.3a, P_{src} is a program written in a source-level language. By applying a compilation scheme, it is translated into a target program P_{tgt}. We use $P_{src} \rightsquigarrow P_{tgt}$ to denote this translation relation. To understand the semantic effect of applying this translation, we lift this relation to the execution graphs that are associated to each of the programs. In this case, let G_{src} be an execution graph of P_{src}. Applying the compilation scheme has the effect of transforming this execution graph to a target-level execution graph, G_{tgt}, which is an execution graph of P_{tgt}. We denote the relation between the two execution graphs as $G_{src} \rightsquigarrow G_{tgt}$. While not all source-level behaviors of a program are necessarily preserved at the target level by a correct compiler, there should not be any *new* behavior at the target level introduced by the compilation process. Following this intuition, if G_{tgt} is consistent under the memory model of the target language, then a correct compilation scheme should ensure that G_{src} is consistent under the memory model of the source language. Note that the memory models of the source and target language may not be the same. Hence, we use two different colors to denote the

$$\begin{tikzcd}
P_{src} \arrow[r, phantom, "\rightsquigarrow"] \arrow[d] & P_{tgt} \arrow[d] \\
G_{src} \arrow[r, phantom, "\rightsquigarrow"] & G_{tgt}
\end{tikzcd}$$

(a) Compilation

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[d] \\
G_{\sigma} \arrow[r, "\text{Predict}"'] & G_{\rho}
\end{tikzcd}$$

(b) Predictive Analyses

$$\begin{tikzcd}
P \arrow[r, phantom, "\rightsquigarrow"] \arrow[d] & P' \arrow[d] \\
G \arrow[r, phantom, "\rightsquigarrow"] & G'
\end{tikzcd}$$

(c) Transformations

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[d] \arrow[r, phantom, "\rightsquigarrow"] & P' \arrow[d] \\
G_{\sigma} \arrow[rr, bend right, "\text{Predict}"'] & G_{\rho} \arrow[r, phantom, "\rightsquigarrow"] & G'_{\rho}
\end{tikzcd}$$

(d) Predictive Analyses with Transformations

Figure 2.3: Relationships of Programs and Executions in Compilation, Prediction, and Transformation

two execution graphs, each being consistent under the respective memory model. In Chapter 3, we formally define the relation between G_{src} and G_{tgt} and use this relation and the axiomatic memory models of the source and target languages to define a correct compilation scheme.

Example From the example in previous section, we see that Volatile mode in Java enforces program orders. Under the hood, there is a compilation scheme that translates the program that we saw from the previous section into the following program in x86 instructions.

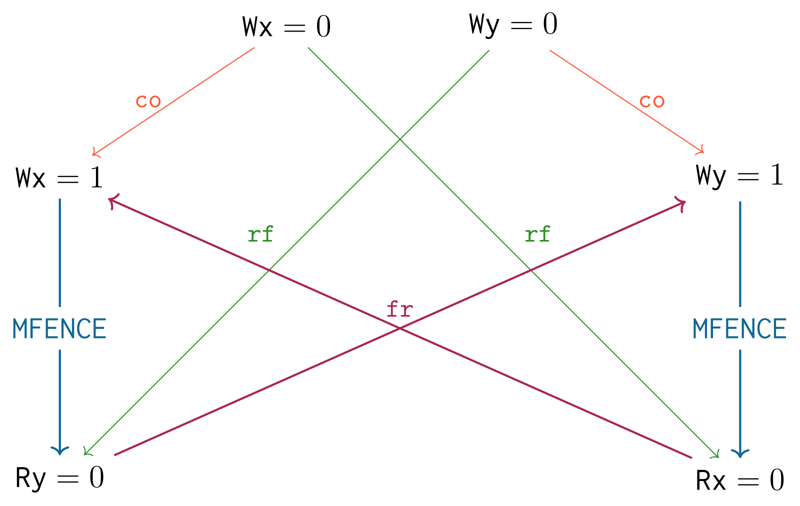


Figure 2.4: Store Buffering (SB) with MFENCE inserted

initially `x == y == 0`	
Thread 1	Thread 2
`MOV [x], 1`	`MOV [y], 1`
`MFENCE`	`MFENCE`
`MOV eax, [y]`	`MOV ebx, [x]`

The Java compiler inserts `MFENCE` instructions when compiling the program to x86 instructions, which results in the execution graph shown in Fig. 2.4. In this execution graph, the inserted `MFENCE` instructions transforms the previous cycle formed by po and `fr` into a cycle formed by `MFENCE` and `fr`.

To show that this compilation is correct, we need to show that G_{src}, which is the forbidden execution graph shown in Fig. 2.2, is still forbidden after being translated to G_{tgt}, which is the execution graph shown in Fig. 2.4. In this case, the execution in Fig 2.4 is forbidden under the x86-TSO model precisely due to the presence of `MFENCE` instructions. Thus, we say the Java program is correctly compiled into a x86 program.

2.4.2 Predictive Analysis

Predictive analysis is a class of dynamic program analyses that finds concurrency bugs by predicting executions. As shown in Fig. 2.3b, let P be a program with a reported bug to be fixed. G_σ is an execution graph of P that is captured by the analysis and is sequentially consistent. Note that predictive analyses can report bugs that are not necessarily present in the recorded execution G_σ, but in some execution of the same program P. In this case, let G_ρ be such a *witness* execution where the bug is present. We denote the relation between the two execution graphs as $G_\sigma \xrightarrow{\text{Predict}} G_\rho$. Then a correct predictive analysis should ensure that G_ρ is indeed a valid execution of P. We identify two important aspect of defining the notion of validity in this context, *executability*, which establish the relation between P and G_ρ, and *memory consistency*, which is determined by the memory model under which P is executed. Note that, although G_σ is captured under sequential consistency, the memory model under which P is expected to be executed is not necessarily sequential consistency. Therefore, we use two different color to distinguish their consistency property. In Chapter 4, we formally define the correctness of predictive analyses and provide a proof recipe supporting the definition.

Example From Chapter 1 we saw a simple program with two threads, each with two instructions that accesses shared memory locations. Now consider the following program, which is an extended version:

initially `x == y == z == 0`	
Thread 1	Thread 2
`x := 1`	`y := 1`
`r1 := y`	`r2 := x`
`if (r1 == 0)`	`if (r2 == 0)`
`z := 1`	`z := 2`

While we know that there are data races on location x and y, is there any data race for location z? Recall that a data race is a pair of memory events from different threads accessing the same memory location such that at least one of the events is a write. In this program, there are two write events for location z, generated from `z := 1` and `z := 2` respectively. Therefore, we need to determine whether the write events generated from these two instructions form a data race. Note that, the `z := 1` and `z := 2` are each enclosed in an `if`-statement in the program. Therefore, the write events are only generated when the `if`-condition is true. In this case, when `r1` and `r2` are both `0`. From Chapter 1, we know that under sequential consistency, at least one of `r1` and `r2` is guaranteed to be `1`. Therefore, under sequential consistency, there is no data race for location z since at least one of the `if`-condition is `false` in every execution. On the other hand, it is possible for `r1` and `r2` to be both `0` under x86-TSO, as we have seen previously. Therefore, it is possible for `z := 1` and `z := 2` to be executed at the same time under x86-TSO. As a result, there is a data race formed precisely by the events generated from these two instructions.

To show that their results are sound, predictive analyses construct a witness execution for each reported bug that dirctly expose the bug. Intuitively, the witness execution consists of an execution prefix that occurs immediately before the bug and the bug itself. In this example, the SB execution shown in Fig 2.5 is such a witness execution for the data race on location z under x86-TSO. The bug is highlighted in 🟨, which immediately occurs after a SB execution. Note that there are two properties that this execution satifies: (1) it is a valid execution (prefix) of the program, and (2) it is consistent under x86-TSO. We call the former property **executability** and the latter property **memory consistency**.

2.4.3 Transformations

Transformations are commonly seen in compilers as well, in addition to compilation. Different from compilation, transformation is an operation that may insert or eliminate memory events or change the threads of which the events are generated from. On the other hand, transformations are typically applied under a single memory consistency model. Therefore, we study the correctness of transformations as a similar but different property than the correctness of compilations. In Fig. 2.3c, let P be a program before applying a transformation and P' be the program obtained

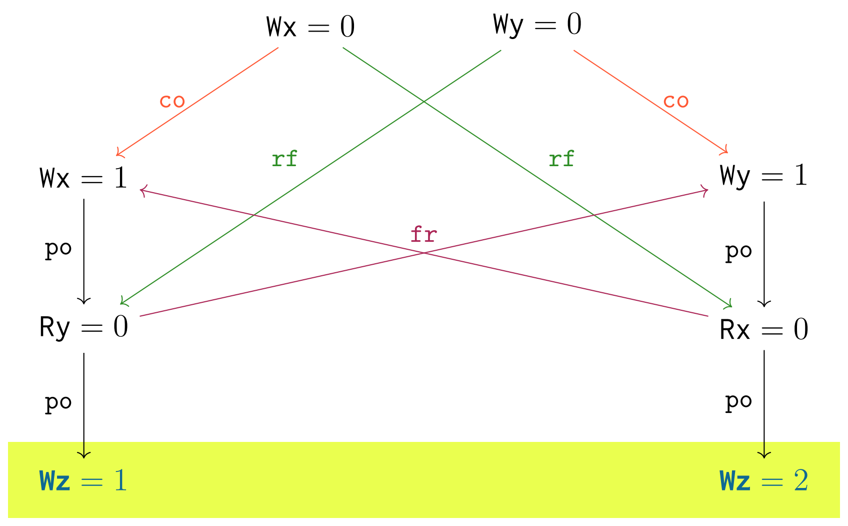


Figure 2.5: Store Buffering (SB) with a data race

after applying the transformation. The effect of the transformation is lifted to execution graphs, hence $G \rightsquigarrow G'$. From the perspective of memory consistency, a correct transformation should not introduce new behavior by turning an inconsistent execution into a consistent one. Therefore, if G' is consistent, then G must be consistent as well. Note that the memory model used to determine their consistency is the same. In the first half of Chapter 5, we formally define this notion of correctness for transformations and prove the correctness of a list of transformations under the memory model of the Java Access Modes.

Example Compilers can reorder instructions during the process of compilation. For the program we saw in Chapter 1, a compiler may transform it into the following program:

initially `x == y == 0`	
Thread 1	Thread 2
`r1 := y`	`y := 1`
`x := 1`	`r2 := x`

The instructions in blue on the first thread are reordered. Note that, after the transformation, it is still possible for both `r1` and `r2` to both be `0` under the model of Java Access Modes (assuming this program is written in Java). This is acceptable since the behavior of the program is already expected before the transformation, as we have discussed in the previous section. On the other hand, the following reordered program is unacceptable:

initially `x == y == 0`	
Thread 1	Thread 2
`r1 = y.getVolatile();`	`y.setVolatile(1);`
`x.setVolatile(1);`	`r2 = x.getVolatile();`

Recall that Volatile mode enforces the program orders among events generated from instructions. The reordering transformation in this example breaks the guarantee of Volatile and introduce a new behavior that is not expected from the source level. Note that now it is possible for `r1` and `r2` to both be `0` under the model of the Java Access Modes after the transformation. In fact, this outcome is allowed for this transformed program even under sequential consistency, which is the strongest memory model. Therefore, this transformation should not be applied to Volatile accesses. We say that the reordering transformation in this example is **sound** if the instructions are regular plain instructions, but **unsound** if they are annotated as Volatile.

2.4.3.1 Transformation for Predictive Analyses

Lastly, just like combining Fig. 2.3a and Fig. 2.3c leads to a definition for a correct compiler, it is also possible to combine Fig. 2.3b and Fig. 2.3c, which provides a way to correctly extend existing predictive analyses, which only focused on predicting bug under sequential consistency, with the capability of predicting bugs under some weak memory models. In Fig. 2.3d, let P be a program with an execution graph G_σ captured under sequential consistency. If the analysis is only able to predict bugs under sequential consistency, then it would not be able to directly predict the bug-exhibiting execution, G_ρ, which is *not* sequentially consistency but is consistent under a weaker memory model. However, we can transform G_ρ into an execution graph G'_ρ that has the same behavior but is sequentially consistent. In other words, if the program behavior of G_ρ can

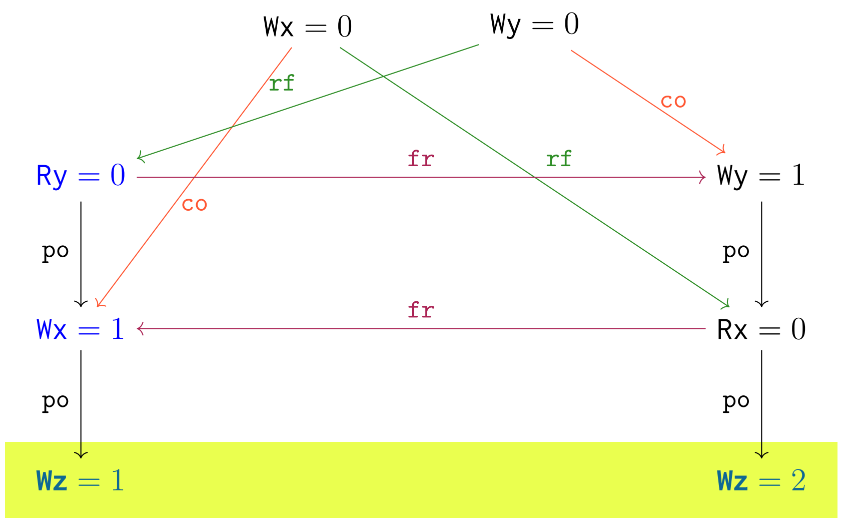


Figure 2.6: A sequentially consistent execution after transformation witnessing the data race

be *explained* by a sequentially consistent execution graph G'_ρ that also witnesses the reported bug, then showing the existence of G'_ρ is sufficient for a correct predictive analysis. However, although there are many sound transformations that support this definition of correctness, not all transformations are helpful for catching more bugs in weak memory. In the second half of Chapter 5, we borrow two transformations for x86-TSO from [LV16] that are previously proved sound and define a new sound transformation for ARMv8. For demonstration, we augment an existing SMT-based predictive analysis with one of the sound transformations for x86-TSO and use the extended algorithm to predict new data races under x86-TSO, which could not be predicted by the original analysis algorithm. Lastly, we prove the correctness of the new algorithm by following the same recipe from Chapter 4.

Example Many existing predictive analyses assumes sequential consistency and construct witness executions with a single global linear order among events. They cannot recognize different types of orders (that we have been using different colors to distinguish in the execution graphs) and do not allow cycles of any form. As a result, they cannot be directly used to predict data races under weak memory models, such as the example we have shown previously.

On the other hand, we have seen that reordering transformation can be applied if the instructions are not annotated as Volatile. The outcome that both `r1` and `r2` are `0` becomes possible under sequential consistency after the transformation. We can leverage this observation and construct a sequentially consistent execution graph with the same outcome as the execution graph from Fig. 2.5 to *represent* the witness of the bug.

As shown in Fig. 2.6, a transformation is applied on the execution graph from Fig. 2.5 and the two events in blue are reordered. Since there is no cycle in this new execution graph, it is a sequentially consistent execution, and hence can be constructed by an existing predictive analysis as a witness for the data race on location z as highlighted. It's important to note that this transformed execution graph is no longer an execution of the original program P, but rather a transformed program P'. The previous correctness properties, i.e., executability and memory consistency, are not satisfied by this transformed execution graph in Fig. 2.6. Instead, properties of this execution graph *imply* the correctness properties of the actual witness execution in Fig. 2.5. Thus, the soundness of the transformation becomes critical to ensure this relationship between the two execution graphs is valid.

CHAPTER 3

Compilation Correctness of Java Access Modes

The content of this chapter was included in a paper published in ECOOP 2022 [LBP22].

In OpenJDK 9, the Java programming language introduced the VarHandle API with Access Modes to provide a standard set of operations that gives clear semantics to programs with shared object fields. Among the four available Access Modes (which we will explain in Section 3.2 in detail), programmers are allowed to use Volatile mode to ensure the consistency of updates on shared variables. Conceptually, the set of Volatile mode accesses in a program is totally ordered [Lea18]. If all of the accesses in a program are in Volatile mode, then the program should only have sequentially consistent executions since all accesses in that program are totally ordered.

Sadly, this basic property of Volatile mode does not hold under the current implementation of the Java compiler in OpenJDK 9 HotSpot JVM. That is, marking all accesses as Volatile in a Java program can still result in behaviors that are not sequentially consistent when compiling to Power [SSA11]. In particular, the C1 and the C2 compilers in HotSpot do not provide enough synchronization between a Volatile read and a Volatile write when compiling to the Power architecture.

While we leave the details of their respective compilation schemes to Section 3.1, when a program includes a sequence of a Volatile read followed by a Volatile write, there is no `hwsync` instruction inserted in-between. Without the `hwsync`, it is possible for threads to disagree on the orders in which instructions are executed. As a consequence, the compilation schemes can still cause violations of sequential consistency in programs with all accesses marked Volatile. We have contacted the maintainers of the OpenJDK about this issue and a bug report has been filed [Shi21].

One solution is to add the missing `hwsync` instruction to restore sequential consistency for Volatile. While the change to the compilation scheme appears to be simple, the work of verifying its soundness is challenging. First, the formal memory model Jam (hereafter Jam19) [BP19] exhibits the same issue as the HotSpot compilers. That is, it cannot guarantee sequential consistency for programs with all accesses marked Volatile. Therefore, we revise the language model to fix this issue. To ensure the change to the model is valid we formally verify its key properties, such as the standard DRF-SC theorem, and leverage a set of empirical litmus tests via our implementation of Java in Herd7 [AMT14] that keeps the model valid. We call the revised model Jam21 to distinguish from the original version. Second, the language model defines the semantics of `fullFence()` with a total order. However, many target-level architectures such as the Power memory model [SSA11] only specify a partial observable order among their synchronization mechanisms (fence cumulativity). Therefore, we develop an intermediate language model, Jam21', to bridge Jam21 with the target level models. We show that Jam21' yields the same observable program executions as Jam21 but does not specify a total order among `fullFence()`s, which simplifies the proof for compilation correctness.

Supplementary Material The proofs of the theorems appear in this paper are available in the appendices (which are available in the full version of the paper). The following are also available as artifact of this paper at `https://github.com/ShuyangLiu/ECOOP22-Supplementary-Material`.

- The extended Herd7 tool suite with the Java architecture (merged to the official repository in 2022 after the paper was published).
- The litmus tests that appear in this chapter.
- The Coq proofs for some of the theorems in this chapter.

3.1 The Problem of Compiling Volatile and How to Fix it

In this section we use an example to demonstrate that the approach implemented by the HotSpot JVM compilers does not provide sequentially consistent semantics even when all accesses use Volatile mode.

Consider the volatile-non-sc.4 example shown as an execution in Fig.3.1. In this example, there are four concurrent threads (`P1`, `P2`, `P3`, and `P4`) accessing two shared integer variables (x and y). The notation `Wx = 1` means "writing to variable x with value 1". The notation `Rx = 0` means "reading from variable x and the value returned is 0". In addition, each variable is initialized to 0 at the beginning before the threads start execution. The small superscript on each memory access denotes the access mode that the access uses. For example, $\texttt{Rx}^{\texttt{V}}$ means "reading with Volatile mode".

If all of the read and write accesses in this program use Volatile mode, would the reads ever return the values that are specified in the figure?

According to the specification [Lea18], the program must exhibit sequentially consistent behavior because all accesses are marked Volatile:

> *"When all accesses use Volatile mode, program execution is sequentially consistent, in which case, for two Volatile mode accesses A and B, it must be that A precedes execution of B, or vice versa."*

Therefore, we are interested in whether the example in Fig. 3.1 is sequentially consistent. Sequential consistency, as first defined by [Lam79], requires a total sequential order that preserves program order and the values returned by the reads are compatible with this total order. Following the definition, the execution in Fig. 3.1 does not satisfy sequential consistency. To see this, we demonstrate a contradiction under the guarantees of sequential consistency. Consider the following order constraints:

1. By program order, we know that (a) occurs before (b).

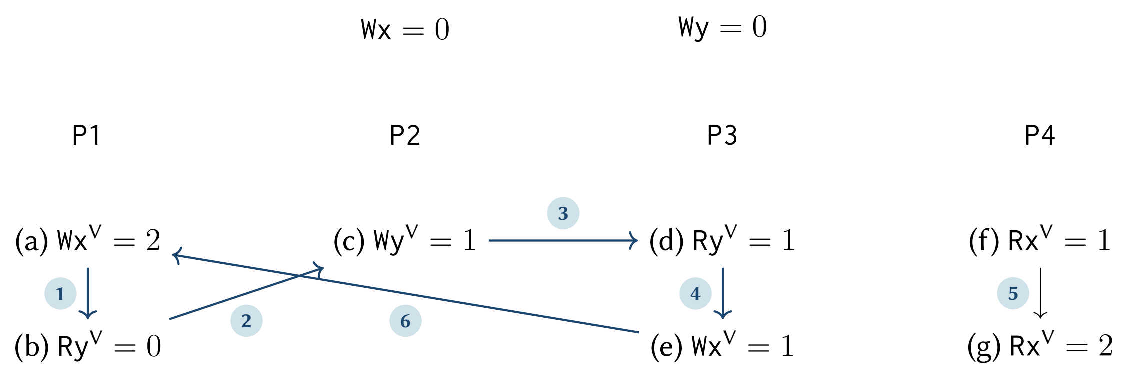


Figure 3.1: volatile-non-sc.4 under the sequential consistency model, Forbidden

2 Since the value (b) gets is the initial value, it must occur before (c) writes to the location y.

3 Then, (d) reads the value written by (c), so (c) occurs before (d).

4 By program order, (d) occurs before (e).

5 By program order, (f) occurs before (g).

6 Now, looking at P4, we know that the value of x changed from 1 to 2. Therefore, we can infer that (e) occurs before (a) since (e) is the only write to x with a value of 1 and (a) is the only write to x with a value of 2.

In this execution, we find a cycle: (a) $\longrightarrow$ (b) $\longrightarrow$ (c) $\longrightarrow$ (d) $\longrightarrow$ (e) $\longrightarrow$ (a) which is highlighted in Fig. 3.1 with the “occurs before” relation represented as edges in the execution graph. Sequential consistency requires an irreflexive total order among all instructions. Therefore, the chain formed by the total order should be acyclic, i.e., a valid execution should not exhibit any cycle in its graph. Thus, this execution is inconsistent under sequential consistency and should be forbidden.

However, despite the promise of sequential consistency given by the source-level Volatile semantics, the compilation scheme found in the Java compilers for Power allows the example execution in Fig. 3.1. To see this, we present the compilation scheme from the C1 compiler which is the more conservative compiler of HotSpot. We then give a Power-consistent execution graph corresponding to the example in Fig. 3.1.

The Power architecture adopts a relaxed memory model and provides fence instructions to recover sequential consistency. Two main types of fence instructions, the stronger fence `hwsync` and the weaker fence `lwsync`, are usually used by the compilers to enforce synchronization guarantees. Using `lwsync` usually gives better performance but the synchronization guarantee of `lwsync` is weaker than `hwsync`. In particular, while both fence instructions carries a set of writes (Group A writes) when propagating to another thread, `lwsync` does not require an acknowledgement to continue executing the instructions after it. On the other hand, a `hwsync` requires an acknowledgment marking that it (along with its Group A writes) has propagated to all threads before proceeding to the next instruction.

The compilation to Power for Volatile accesses on C1 is the following [1]:

$$\mathtt{R}^{\mathtt{V}} \rightsquigarrow \mathtt{hwsync}\ ;\ \mathtt{lwz}\ ;\ \mathtt{lwsync}$$

$$\mathtt{W}^{\mathtt{V}} \rightsquigarrow \mathtt{lwsync}\ ;\ \mathtt{stw}\ ;\ \mathtt{hwsync}$$

A Volatile read is compiled to a `hwsync` instruction followed by a load instruction and a `lwsync` instruction; a Volatile write is compiled to a `lwsync` instruction followed by a store instruction and a `hwsync` instruction.

Fig. 3.2 shows the example from Fig. 3.1 according the compilation scheme in the C1 compiler[2].

[1]This compilation scheme was found in the OpenJDK 13 HotSpot compiler and it follows from a previously inaccurate description in the documentation [Lea18] regarding the semantics of Volatile accesses. We have contacted the author and the documentation has been corrected in the latest version while the compiler bug (although reported) is still not fixed at the time of writing.

[2]The C2 compiler yields a slightly different compilation scheme for Volatile reads: Instead of inserting a `lwsync` fence after the load instruction, it emits a control dependency followed by an `isync` instruction, which we denote as `ctrlisync`. But in this example, the resulting execution graph is effectively the same as C1's because the effect of `ctrlisync` is subsumed into the `lwsync` or the `hwsync` instruction that it follows. In addition, we have simplified the compiled code (such as eliminating the fence instructions at the beginning or end of the threads and merging consecutive fence instructions) without changing its semantics for clarity here.

	$\mathtt{Wx} = 0$	$\mathtt{Wy} = 0$	
P1	P2	P3	P4
(a) $\mathtt{Wx}^{\mathsf{V}} = 2$	(c) $\mathtt{Wy}^{\mathsf{V}} = 1$	(d) $\mathtt{Ry}^{\mathsf{V}} = 1$	(f) $\mathtt{Rx}^{\mathsf{V}} = 1$
(B1) `hwsync`		(B2) `lwsync`	(B3) `hwsync`
(b) $\mathtt{Ry}^{\mathsf{V}} = 0$		(e) $\mathtt{Wx}^{\mathsf{V}} = 1$	(g) $\mathtt{Rx}^{\mathsf{V}} = 2$

Figure 3.2: volatile-non-sc.4.ppc translated to Power by HotSpot C1, Allowed

The Power memory model [SSA11] allows the behavior annotated in Fig. 3.2. The full trace of the execution can be found in Appendix K.5. Here we give a brief explanation. First note that a write operation is split into multiple steps and can be propagated to foreign threads in different orders if not properly synchronized. Furthermore, the `lwsync` in `P3` is not sufficient in this case. In particular, the `lwsync` does not require an acknowledgement before proceeding to the next instructions and it only requires (c) `Wy = 1` to be propagated when itself needs to be propagated to the thread (the cumulativity of `lwsync`). Since `P4` needs to read from (e) `Wx = 1`, which is subsequent to (B2), (B2) needs to be propagated to `P4` before (e) `Wx = 1` is propagated to `P4`. The propagation of (B2) `lwsync` makes sure that (c) `Wy = 1` is propagated to `P4` before it can read x (even though it doesn't really need to read the value of y). On the other hand, `P1` does not have any instructions reads from an instruction of `P3` that comes after (in program order) (B2). Therefore, it does not require (c) and (B2) to be propagated to it when it executes (b). As a result, (c) can be propagated to `P1` long after reaching `P3` and hence letting `P3` and `P1` have different views of the memory during the execution. When `P1` tries to read the value of y, it can only get an initial value of 0 since the newer value has not been propagated to `P1` yet. Consequently, this non-SC execution is allowed (consistent) under the Power memory model, despite that the semantics of the "all-Volatile" source program requires it to be forbidden.

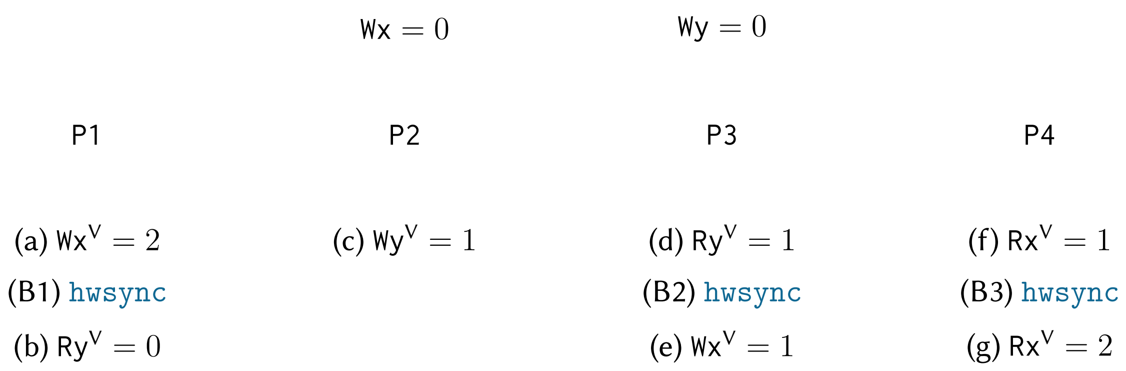

$\mathtt{Wx} = 0$
$\mathtt{Wy} = 0$
P1
P2
P3
P4
(a) $\mathtt{Wx}^{\mathsf{V}} = 2$
(B1) hwsync
(b) $\mathtt{Ry}^{\mathsf{V}} = 0$
(c) $\mathtt{Wy}^{\mathsf{V}} = 1$
(d) $\mathtt{Ry}^{\mathsf{V}} = 1$
(B2) hwsync
(e) $\mathtt{Wx}^{\mathsf{V}} = 1$
(f) $\mathtt{Rx}^{\mathsf{V}} = 1$
(B3) hwsync
(g) $\mathtt{Rx}^{\mathsf{V}} = 2$

Figure 3.3: volatile-non-sc.4.ppc translated to Power using the revised compilation scheme, Forbidden

The solution to fix this issue is quite straightforward. Instead of letting Volatile read be translated using "leading fence" while Volatile write be translated using "trailing fence", they should both use the same fence inserting strategy (both leading fence or both trailing fence).[3] Therefore, the correct compiler scheme for Volatile should be:

$$\mathtt{R}^{\mathtt{V}} \rightsquigarrow \mathtt{hwsync\ ;\ lwz\ ;\ lwsync}$$

$$\mathtt{W}^{\mathtt{V}} \rightsquigarrow \mathtt{hwsync\ ;\ stw}$$

With the revised compilation scheme we can demonstrate that the example of Fig. 3.1 is forbidden in accordance with the required SC semantics. The resulting execution graph is shown in Fig. 3.3. While most of this example matches Fig. 3.2, (B2) now is a `hwsync` instruction. As an effect of this change, (B2) is now required to be propagated to every thread and get acknowledged before start executing (e). As a result, at the time when (c) is propagated to P4 (as a result of the cumulative effect of (B2) just like in Section. 3.1), it must also have propagated to P1 due to the

[3]Here we choose to show the leading fence strategy for simplicity. However, the trailing fence strategy is symmetric to leading fence and the same correctness proof works for both conventions given it's used consistently (more details can be found in Section 3.4.1). In practice, it is usually preferable to use trailing fence strategy for better performance.

acknowledgement required by the `hwsync` at (B2). Therefore, it becomes impossible for (b) to read the value 0 because Power requires reads to always read from the latest value that has been propagated to the thread. That is, this execution is now forbidden by Power, aligning with the sequentially consistent semantics promised by the Java Volatile mode. Note that the reasoning is the same if we use a "trailing fence" scheme. The key is to deploy a fence insertion strategy such that there is a `hwsync` fence inserted between every pair of Volatile accesses.

Interestingly, we found similar compilation schemes applied to other architectures in HotSpot as well. This is not an accident. The source of this compiling behavior stems from the IR phase of the compiler. At the IR (called the Ideal Graph IR in HotSpot) level, a Volatile read is translated to a `fullFence()` followed by an Acquire read; a Volatile write is translated to a Release write followed by a `fullFence()`. Then each compiler back end translates the code further using the corresponding template file that maps the IR to specific architecture instructions. In the case of Power, a `fullFence()` is mapped to the `hwsync` instruction and Release-Acquire accesses are implemented using the `lwsync` instruction. While the example we provide here focuses on the compilation to Power, the more fundamental issue here is a lack of `fullFence()` between a Volatile read and a Volatile write at the IR encoding level. JAM19 aligns with this encoding when specifying the semantics of Volatile memory operations. As a result, JAM19 also exhibits the same problem. That is, when all memory accesses are Volatile, JAM19 does not guarantee sequential consistency.

3.2 Formal Model

In this section we present the revised model JAM21, which we use as our theoretical foundation for proving compiler correctness in the rest of the paper. We begin by introducing the basic syntax (Section 4.3) used in the rest of the paper. Then we give the formal definition of JAM21 in Section 3.2.2.

3.2.1 Basic Syntax

We adopt the syntax of [BP19] and the `cat` language [AMT14] in addition to some utility functions.

Given a program $P \in$ Prog, there is a set of execution graphs associated with P, which we use G to denote a single execution graph. Each execution graph consists of sets of memory events generated from P. In particular: G.Evts denotes the set of memory events in G; $G.\texttt{F}$ denotes the set of fence events in G; $G.\texttt{IW}$ denotes the set of initialization writes of G; $G.\texttt{FW}$ denotes the set of final writes of G; G.Wrts denotes the set of write events in G; G.Rds denotes the set of read events in G; and $G.\texttt{RMW}$ denotes the set of read-modify-write events in G. Note that we treat each RMW event as a single event such that $G.\texttt{RMW} \subseteq G$.Wrts and $G.\texttt{RMW} \subseteq G$.Rds. In addition, for RMW operations such as *compare-and-swap* (CAS), we assume the operation is on its success comparison path. They are sometimes implemented using LL/SC instructions on hardware, which cannot guarantee atomicity if the comparison fails. We assume each write event to the same memory location has an unique value for simplicity.

For each memory event $e \in G$.Evts, we define the following attributes: $G.AccessMode(e)$ is the Access Mode of event e in G; $G.val(e)$ is the value of event e in G; $G.loc(e)$ is the memory location of event e in G; and $G.tid(e)$ is the thread identifier from which e is executed. Finally, we use the symbol $\mathcal{G}$ to denote the set of all execution graphs.

The memory events in each G are related by order relations.

- The program order (po) is a partial order relation (po $\subseteq G$.Evts $\times$ G.Evts) specified by P. We use the notation $i_1 \xrightarrow{\text{po}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by po and G.po to denote the set of all pairs relates by po in G.
- The reads-from (`rf`) order is a partial order relation (`rf` $\subseteq G$.Wrts $\times$ G.Rds). For each read event i_2, there exists a unique write event i_1 such that $G.val(i_1) = G.val(i_2)$ and $G.loc(i_1) = G.loc(i_2)$. We use the notation $i_1 \xrightarrow{\texttt{rf}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by `rf` and $G.\texttt{rf}$ to denote the set of all pairs relates by `rf` in G.

- The trace order (`to`) is a total order among all events in an execution graph.
- Model-Specific relations. There are sets of relations that are specifically defined by the memory model. They are derived from the event attributes, po, and `rf` using the semantic rules of the memory model. We will detail them in the next few sections. We use the notation $i_1 \xrightarrow{\mathtt{R}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle \in G.\mathtt{R}$.

We also use operations on relations: given relations $\mathtt{R}_1$ and $\mathtt{R}_2$, we use composition $\mathtt{R}_1 ; \mathtt{R}_2$, union $\mathtt{R}_1 \,|\, \mathtt{R}_2$, intersection $\mathtt{R}_1 \,\&\, \mathtt{R}_2$, complement $\sim\mathtt{R}_1$, transitive closure $\mathtt{R}_1^+$, identity relations over a set $[A]$, and inversion $\mathtt{R}_1^{-1}$.

Lastly, we use the notation acyclic($\xrightarrow{\mathtt{R}}$) to denote that R is acyclic in the execution history.

3.2.2 The Jam21 Model

In this section, we present the Jam21 model. The full definition of Jam21 written in the `cat` language [AMT14] can be found in Appendix A. We explain several excerpts of the formal model.

There are five available access modes in Jam21: Plain mode, Opaque mode, Release mode, Acquire mode, and Volatile mode. The synchronization effect of the access modes are partially ordered using $\sqsubseteq$:

$$\mathsf{Plain} \sqsubseteq \mathsf{Opaque} \sqsubseteq \{\mathsf{Release}, \mathsf{Acquire}\} \sqsubseteq \mathsf{Volatile}.$$

Among the five access modes, the Plain mode is the most relaxed mode that imposes the lease synchronization whereas the Volatile is the strongest mode that imposes the most synchronization effect. In particular, the synchronization effect of each mode are expressed in terms of the *visibility orders*, which we explain in details next.

3.2.2.1 Visibility

At the center of JAM21 is the notion of *visibility orders* (vo). The most basic form of visibility, vo includes the reads-from (rf) relation. Intuitively, a write has to be *visible* to the read that read from it. In other cases, visibility comes from the synchronization effects of various access modes. Both Volatile (V) and Release(REL)-Acquire(ACQ), (RA as the union) accesses provide visibility effect. Note that Volatile accesses are also included in the set of accesses that are considered Release-Acquire by the model since their semantics are defined in a cumulative style. The vo order can be derived from ra and svo orders, which captures the synchronization effects of Release-Acquire memory events or fences, or spush and volint orders, which capture the synchronization effects of Volatile memory events or `fullFence()`s. The union of the two orders is the push order. In addition, the pushto order is a subset of trace order (to) restricted to Volatile memory events and `fullFence()`s. Composing pushto with push emulates the cross-thread total order among Volatile events and `fullFence()`s, which is also part of the vo order. Finally, po restricted to the same memory location (po-loc) is also included as part of the vo definition, since JAM21 respects the SC semantics per memory location.

$$
\begin{aligned}
\texttt{ra} &\triangleq \texttt{po ; [REL | V] | [ACQ | V] ; po} \\
\texttt{svo} &\triangleq \texttt{po ; [F \& REL] ; po ; [Wrts] | [Rds] ; po ; [F \& ACQ] ; po} \\
\texttt{spush} &\triangleq \texttt{po ; [F \& V] ; po} \\
\texttt{volint} &\triangleq \texttt{[V] ; po ; [V]} \\
\texttt{push} &\triangleq \texttt{spush | volint} \\
\texttt{vvo} &\triangleq \texttt{rf | svo | ra | push | pushto ; push} \\
\texttt{vo} &\triangleq \texttt{vvo}^{+}\texttt{ | po-loc}
\end{aligned}
$$

Note that the definition of volint has been corrected from JAM19 to ensure sequential consistency for Volatile.

3.2.2.2 Coherence

The coherence order, co-jom, is an order among Opaque or stronger mode writes to the same location. Coherence order edges can be derived using the vo order and the po order among memory accesses.

$$\begin{aligned}
\texttt{WWco(R)} &\triangleq \{\langle i_1, i_2\rangle \mid \langle i_1, i_2\rangle \in G.\mathrm{R} \wedge i_1, i_2 \in G.\mathsf{Wrts} \wedge G.loc(i_1) = G.loc(i_2) \wedge i_1 \neq i_2\} \\
\texttt{coww} &\triangleq \texttt{WWco(vo)} \\
\texttt{cowr} &\triangleq \texttt{WWco(vo ; rf}^{-1}\texttt{)} \\
\texttt{corw} &\triangleq \texttt{WWco(vo ; po)} \\
\texttt{corr} &\triangleq \texttt{[O | RA | V] ; WWco(rf ; po ; rf}^{-1}\texttt{) ; [O | RA | V]} \\
\texttt{co-jom} &\triangleq \texttt{coww | cowr | corw | corr}
\end{aligned}$$

Note that co-jom is different from the definition of co in other memory models such as Power and x86-TSO. Instead of enumerating all possible total coherence order to check the consistency of a given execution history, JAM21 derives coherence order co-jom among memory events from their known relations. Therefore, co-jom is a partial order among writes to the same location in JAM21. We use the notation $i_1 \xrightarrow{\texttt{co-jom}} i_2$ to denote the pair of events $\langle i_1, i_2\rangle$ related by co-jom and G.co-jom to denote the set of all pairs relates by co-jom in G. We use the simpler name co to denote co-jom when the context is clear.

Different from JAM19, Plain mode reads to the same location ordered by po can be reordered by compiler and therefore cannot be used to derive co-jom order.

3.2.2.3 Execution Consistency

Axiomatic models define program semantics as the set of allowed executions. We adopt the same definition of *candidate execution* from [AMT14]. Given a program P and a memory model $\mathcal{M}$, an execution graph G is a *$\mathcal{M}$-consistent execution of P* if G is a candidate execution of P (specified by the architecture of the programming language of which P is written in), and G is $\mathcal{M}$-consistent.

We denote the set of all $\mathcal{M}$-consistent candidate executions of P by $[\![P]\!]_{\mathcal{M}}$.

We now have all the definitions needed to define execution consistency under Jam21.

▶ Definition 1 (Jam21-consistency)
A well-formed execution graph G is Jam21-*consistent* if it satisfies the following two requirements:

1. No-Thin-Air: po | rf is acyclic. acyclic($\xrightarrow{\texttt{po | rf}}$)
2. Coherence.: co-jom is acyclic, acyclic($\xrightarrow{\texttt{co-jom}}$)

We say such an execution history G is *allowed* by Jam21. Otherwise, it is *forbidden*.

For the Jam21 model, we use $[\![P]\!]_{\text{Jam21}}$ to denote the set of all Jam21-consistent execution graphs of P.

Jam21 satisfies a set of properties such as the DRF-SC Theorem. We show the theorems and the proofs in Appendix H and Appendix I.

3.2.2.4 Validation of Jam21 with Litmus Tests

The experimental validation of the Jam21 model includes two parts.

First, we implement the Java *architecture* in Herd7. Herd7 [AMT14] was developed to simulate program executions with user-defined memory models. An *architecture* in Herd7 provides the parser for litmus tests written in the language corresponding to the architecture and an operational semantics of the instructions that appear in litmus tests. Herd7 uses the parser and the instruction semantics from the architecture to form an internal representation of the input litmus test and generate the set of all possible executions. Then, Herd7 checks the consistency of the executions using memory models written in the cat language. As of today, several mainstream architectures, such as C/C++11 [LVK17], x86 [OSS09], ARM [PFD18], and Power [SSA11], have been implemented and included in Herd7's official repository. Unfortunately, Java is not. Jam19 [BP19] validated its formalization by mapping memory events to other architectures' events that exists in the Herd7 repository and run the litmus tests in the architecture's language. The mapping roughly captures part of the compilation scheme but it is neither complete nor proven sound. For example, in its mapping to ARMv8, Volatile accesses are ignored and not mapped to any memory event. Hence this approach is invalid and the results cannot be trusted though they show intentions on how JAM19 was expected to behave. Therefore, we extend the HERD7 tool suite with the Java architecture and translate the set of litmus tests used for testing JAM19 to Java[4]. A detailed description of each supported instruction is shown in Appendix K.1.

Second, we validate the JAM21 model using the Java translation of the set of litmus tests that was originally used to validate JAM19 and compare their outcomes. The results are mostly the same as the results from JAM19 except for three cases that are relevant to the inconsistency issue discussed earlier in this paper because we wish to fix the issue while keeping other parts of the model unchanged. The three exceptions reveal another aspect of the change, accommodating both the leading fence convention and the trailing fence convention, whereas JAM19 forced the compiler to choose a particular (problematic) convention. Since the compiler is free to choose either convention, a full synchronisation is only guaranteed to appear between a pair of Volatile accesses. In effect, certain executions that was forbidden by JAM19 are allowed by JAM21 since it is no longer guaranteed that Volatile writes are *followed* by a full synchronisation and Volatile reads are *prepended* with a full synchronisation. In addition, we have added new litmus tests for showing the change in the semantics of Volatile, volatile-non-sc.4 and volatile-non-sc.5. While JAM19 allows the non-sequentially consistent behavior, JAM21 correctly forbids them. We further translated the examples to Power using the problematic compilation scheme, volatile-non-sc.4.ppc and volatile-non-sc.5.ppc, and the tests are indeed allowed by the Power memory model. Please see Appendix K.2 for a detailed report.

[4]Note that not all tests are translatable. For example, for the cases that test address dependencies, there is no corresponding Java version since the notion of address dependency does not exist in Java. We drop a small set of litmus tests due to this reason.

3.2.3 The JAM21' Model

JAM21 enforces a total order among `fullFence()`s, which introduces complexity when proving compilation correctness. Therefore, we introduce an intermediate memory model JAM21' that is *observationally equivalent* to the JAM21. We start by defining and proving the observational equivalence between the two models. Then we use JAM21' to prove the correctness of the compilation schemes.

The JAM21' model is the same as JAM21 except for the semantics of full fences. Instead of having a total order on full fences, JAM21' only enforces order when there is a communication edge. The full semantics of JAM21' can be found in Appendix B. Here we only include the updated portion. The definition for `chapo` is newly added[5]. The cross-thread synchronization effect of `fullFence()`s is then defined as `push`; `chapo`; `push` (instead of `pushto`; `push` as in JAM21):

$$\texttt{chapo} \triangleq \texttt{rfe} \,|\, \texttt{fre} \,|\, \texttt{coe} \,|\, (\texttt{fre} \;;\; \texttt{rfe}) \,|\, (\texttt{coe} \;;\; \texttt{rfe})$$

$$\texttt{vvo} \triangleq \ldots \,|\, \texttt{push} \;;\; \texttt{chapo} \;;\; \texttt{push}$$

The rest of JAM21' are the same as JAM21.

▶ DEFINITION 2 (JAM21' CONSISTENCY)
A well-formed execution graph G is JAM21'-*consistent* if it satisfies the following two requirements:

1. NO-THIN-AIR: po | `rf` is acyclic. acyclic($\xrightarrow{\texttt{po}\,|\,\texttt{rf}}$)
2. COHERENCE: `co-jom` is acyclic, acyclic($\xrightarrow{\texttt{co-jom}}$)

We say such an execution history G is *allowed* by JAM21'. Otherwise, it is *forbidden*.

For the JAM21 model, we use $[\![P]\!]_{\text{JAM21'}}$ to denote the set of all JAM21'-consistent execution histories of P.

[5]The name `chapo` comes directly from the Power Memory Model in the Herd [AMT14] repository. We use the same name here so that the readers can easily relate them

It is important to note that JAM21' is *observationally equivalent* to JAM21, which means they allow the same visible program behaviors given the same program. Intuitively, each consistent execution under JAM21 has a corresponding consistent execution under JAM21' with the same set of events and the same observable value on each event. Formally, we give the following definitions for observational equivalence.

▶ DEFINITION 3 (OBSERVATIONAL EQUIVALENCE OF EXECUTION GRAPHS)
Let G and G' be two well-formed execution graphs. We say G and G' are **observationally equivalent**, writes $G \doteqdot G'$, if:

- G.`IW` = G'.`IW`, G.`FW` = G'.`FW`, and G.Evts = G'.Evts
- G.po = G'.po and G.`rf` = G'.`rf`
- $\forall e \in G.\mathsf{Evts}, G.AccessMode(e) = G'.AccessMode(e)$

The above definition of observational equivalence can be lifted to define an equivalence relation over memory models.

▶ DEFINITION 4 (OBSERVATIONAL EQUIVALENCE OF MEMORY MODELS)
Given a program P, let M_1 and M_2 be two memory models that both support the architecture of P. Let $[\![P]\!]_{M_1}$ be the set of all M_1-consistent executions of P, and $[\![P]\!]_{M_2}$ be the set of all M_2-consistent executions of P. We say M_1 and M_2 are **observationally equivalent**, writes $M_1 \doteqdot M_2$, if:

- $(\Rightarrow)$ For each $G_1 \in [\![P]\!]_{M_1}$, there exists $G_2 \in [\![P]\!]_{M_2}$ such that $G_1 \doteqdot G_2$.
- $(\Leftarrow)$ For each $G_2 \in [\![P]\!]_{M_2}$, there exists $G_1 \in [\![P]\!]_{M_1}$ such that $G_1 \doteqdot G_2$.

We now show the observational equivalence between JAM21 and JAM21'.

▶ THEOREM 1
JAM21$'$ $\doteqdot$ JAM21.

The proof can be found in Appendix D.

3.3 Compilation Correctness: an Overview

While compilation correctness, in general, is a very challenging topic being studied in the research field for many decades, we focus our attention on prove the correctness of *compilation schemes* of concurrent instructions in this chapter. A compilation scheme is a mapping from source-level language instructions to target-level language instructions. When a compiler translates a program from a source-level language to a target-level language, each source-level statement is typically mapped to a sequence of target-level instructions. This translation should preserve the basic semantics of the statement and should not introduce unexpected behaviors. While this mapping may remove statements or insert instructions, we require that the events that are generated at the source level are all preserved at the target level. While modern compilers may also eliminate or insert events for optimizations, we consider those operations as *transformations* and defer the discussion of transformations until Chapter 5.

To better understand the relationship of programs and executions between the source and the target languages of a correct compilation scheme, consider Fig. 3.4. In this diagram, P_{tgt} is a target-level program produced by applying a compilation scheme on a source-level program P_{src}, denoted by $P_{src} \rightsquigarrow P_{tgt}$. P_{src} and P_{tgt} are each associated with a set of execution graphs, which capture their run-time semantics respectively. We use a relation called $CompilesTo$ to associate the two sets of executions, which captures the effect of applying the compilation scheme at the execution graph level. We write $G_{src} \rightsquigarrow G_{tgt}$ for such a pair of execution graphs, where G_{src} is a candidate execution of P_{src} and G_{tgt} is a candidate execution of P_{tgt}.

A correct compilation scheme should not introduce new behavior. In the context of memory consistency models, it means *each forbidden source-level candidate executions should still be forbidden at the target level after the compilation.* Specifically, if G_{src} represent a forbidden program behavior by the source-level memory model, then G_{tgt} should still be forbidden by the target-level memory model. In other words, if G_{tgt} is consistent under the target-level memory model, then G_{src} should also be consistent under the source-level memory model. Formally, we adopt

$$\begin{tikzcd}
P_{src} \arrow[r, phantom, "\rightsquigarrow"] \arrow[d] & P_{tgt} \arrow[d] \\
G_{src} \arrow[r, phantom, "\leadsto"] & G_{tgt}
\end{tikzcd}$$

Figure 3.4: The relationship of programs and executions

the following definition for a correct compilation scheme.

▶ Definition 5 (Compilation Correctness)
Let P_{src} be a source program, M_{src} be a source-level memory model, P_{tgt} be the target program such that $P_{src} \rightsquigarrow P_{tgt}$ with a compilation scheme, and M_{tgt} be a target-level memory model. We say a compiler that compiles P_{src} to P_{tgt} is **correct** if for each $G_{tgt} \in [\![P_{tgt}]\!]_{M_{tgt}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{M_{src}}$ such that $G_{src} \leadsto G_{tgt}$.

In the following two sections, we prove the correctness of the compilation scheme from the Java Access Modes to Power and to x86-TSO, respectively. In each of the two section, the following structure is used:

- We first introduce the formal definition of the target-level memory model.
- We then provide a compilation scheme ($\rightsquigarrow$) that maps each statement of Java Access Modes to a sequence of instructions in the target-level instruction sets.
- We define a $CompilesTo$ ($\leadsto$) relation that reflects the compilation scheme at the execution graph level for compilation to the target language.
- Lastly, we prove the compilation correctness theorems by showing that a memory consistent G_{tgt} implies that G_{src} is also memory consistent.

3.4 Compilation Correctness to Power

In this section, we show that the revised compilation scheme for Power is correct with respect to the Power memory model [SSA11]. We use JAM21' to prove that the revised compilation scheme to Power is correct, since it has been shown to be observationally equivalent to JAM21.

3.4.1 The Power Memory Model

We use the Power memory model defined in HERD7 [AMT14], which consists of the following basic order definitions (Please see Appendix C for the full semantics):

- po and `rf` follows the same definitions as in JAM21 (as described in Section. 3.2).
- `co` is the union of total orders among writes to the same location. Additionally, if i_1 and i_2 are events on different threads and $i_1 \xrightarrow{\text{co}} i_2$, then $i_1 \xrightarrow{\text{coe}} i_2$.
- `ctrl` is the control dependency between memory accesses.
- ppo is the set of preserved program orders. The detailed definition can be found in Appendix C.
- $\texttt{chapo} \triangleq \texttt{rfe} \;|\; \texttt{fre} \;|\; \texttt{coe} \;|\; (\texttt{fre} \;;\; \texttt{rfe}) \;|\; (\texttt{coe} \;;\; \texttt{rfe})$
- $\texttt{com} \triangleq \texttt{rf} \;|\; \texttt{fr} \;|\; \texttt{co}$
- `po-loc` is a subset of po that relates accesses to the same locations.
- `rmw` relates the read and the write access from the same `RMW` memory event.
- $\texttt{hb} \triangleq \texttt{ppo} \;|\; (\texttt{sync} \;|\; \texttt{lwsync}) \;|\; \texttt{rfe}$
- $\texttt{propbase} \triangleq ((\texttt{sync} \;|\; \texttt{lwsync}) \;|\; (\texttt{rfe} \;;\; (\texttt{sync} \;|\; \texttt{lwsync}))) \;;\; \texttt{hb}^*$
- $\texttt{prop} \triangleq \texttt{propbase} \,\&\, (\text{Wrts} * \text{Wrts}) \;|\; (\texttt{chapo}? \;;\; \texttt{propbase}^* \;;\; \texttt{sync} \;;\; \texttt{hb}^*)$
- Additional order definitions can be found in Appendix C.

▶ DEFINITION 6 (POWER CONSISTENCY)
A well-formed execution graph G is Power-*consistent* if it satisfies the following six requirements:

1. SC-PER-LOCATION: (`po-loc | com`) is acyclic.
2. ATOMICITY: `rmw` & (`fre` ; `coe`) is empty.
3. NO-THIN-AIR: `hb` is acyclic.
4. PROPAGATION: (`co` | `prop`) is acyclic.
5. OBSERVATION: (`fre ; prop ; hb`*) is irreflexive.
6. SCXX: `co` | (`po` & (`X` $*$ `X`)) is acyclic (where `X` denotes atomic accesses)

We say such an execution history G is *allowed* by Power. Otherwise, it is *forbidden*.

3.4.2 Compilation Scheme

We use the compilation scheme in Fig. 3.5. Note that this is slightly different from the compilation scheme found in OpenJDK HotSpot compiler in that each Opaque mode read is translated to a load instruction followed by a conditional branch. This enables us to ensure the NO-THIN-AIR property as it is not guaranteed in the Power memory model. The Out-of-Thin-Air (OOTA) problem in axiomatic models has been an active research area for a long time and there exists various ways to use weaker compilation schemes while still ruling out thin-air reads. However, it is out of the scope of this thesis and here we adopt the stronger scheme for Opaque mode. Additionally, we fix the compilation scheme for Volatile as suggested in Section 3.1. Note that both leading fence and trailing fence conventions ensure a `hwsync` instruction is inserted between each pair of Volatile mode accesses as long as they are used consistently (use the same convention for Volatile writes and reads). Therefore, the proof for the trailing fence convention can be carried out in a very similar way as the proof for the leading fence convention.

We start our proof by defining a *CompilesTo* ($\rightsquigarrow$) relation over execution graphs that relates source level executions to target level executions. Intuitively, the process of compilation can

```
         getOpaque() ⇝ lwz ; cmp ; bc
         setOpaque() ⇝ stw
        getAcquire() ⇝ lwz ; lwsync
        setRelease() ⇝ lwsync ; stw
       getVolatile() ⇝ hwsync ; lwz ; lwsync
   (Or getVolatile() ⇝ lwz ; hwsync for trailing fence convention)
       setVolatile() ⇝ hwsync ; stw
   (Or setVolatile() ⇝ lwsync ; stw ; hwsync for trailing fence convention)
      AcquireFence() ⇝ lwsync
      ReleaseFence() ⇝ lwsync
         fullFence() ⇝ hwsync
         getAndAdd() ⇝ hwsync ; _1: ldarx ; add ; stdcx. ; bne _1 ; lwsync
     (Or getAndAdd() ⇝ lwsync ; _1: ldarx ; add ; stdcx. ; bne _1 ; hwsync for trailing fence convention)
  getAndAddAcquire() ⇝ _1: ldarx ; add ; stdcx. ; bne _1 ; lwsync
  getAndAddRelease() ⇝ lwsync ; _1: ldarx ; add ; stdcx. ; bne _1
```

Figure 3.5: Compilation to Power

be seen as a transformation function on executions from source level to target level. With the $\rightsquigarrow$ relation, we can characterize a subset of target level executions constructed via compilation, following a given compilation scheme, from the source level. Note that at this step we do not check whether the resulting execution is consistent under the target level memory model, since the consistency check of an execution comes after the execution is constructed for axiomatic memory models.

▶ Definition 7 (Java CompilesTo Power)
Given a Java program P_{src}, let P_{tgt} be the target-level program compiled from P_{src} using the compilation scheme in Fig. 3.5 (using the leading fence convention). Let G_{src} be a candidate execution graph of P_{src} and G_{tgt} be a candidate execution graph of P_{tgt}. We say $G_{src} \rightsquigarrow G_{tgt}$ if:

- $G_{tgt}.\mathtt{IW} = G_{src}.\mathtt{IW}$ and $G_{tgt}.\mathtt{FW} = G_{src}.\mathtt{FW}$
- $G_{tgt}.\mathsf{Evts} \setminus G_{tgt}.\mathsf{F} = G_{src}.\mathsf{Evts} \setminus G_{src}.\mathsf{F}$, $G_{tgt}.\mathtt{rf} = G_{src}.\mathtt{rf}$, and $G_{src}.\mathsf{po} \subseteq G_{tgt}.\mathsf{po}$
- $G_{tgt}.\mathtt{co} \subseteq G_{src}.\mathtt{to}$
- If $i_1 \in G_{src}.\mathsf{Evts}$, $i_{rmw} \in G_{src}.\mathtt{RMW}$ and $i_{rmw} \xrightarrow{\mathtt{po}} i_1$, then $i_{rmw} \xrightarrow{\mathtt{ctrl}} i_1$ in G_{tgt}

- If $i_R^{\sqsupseteq O} \in G_{src}.\text{Rds}$, $i_1 \in G_{src}.\text{Evts}$ and $i_R^{\sqsupseteq O} \xrightarrow{\texttt{po}} i_1$, then $i_R \xrightarrow{\texttt{ctrl}} i_1$ in G_{tgt}
- If $i_1, i_2 \in G_{src}.\text{Evts}$ and $i_1 \xrightarrow{\texttt{push}} i_2$, then $i_1 \xrightarrow{\texttt{sync}} i_2$ for $i_1, i_2 \in G_{tgt}.\text{Evts}$
- If $i_1, i_2 \in G_{src}.\text{Evts}$ and $i_1 \xrightarrow{\texttt{ra}} i_2$, then $i_1 \xrightarrow{\texttt{lwsync}} i_2$ for $i_1, i_2 \in G_{tgt}.\text{Evts}$

Once we have the source level and target level execution graphs, we use the memory model to check for consistency and show the correctness of the compilation scheme according to Def. 5.

3.4.3 Proof of Compilation Correctness

Now we prove the compilation correctness from JAM21' to Power.

▶ LEMMA 1 (JAM21' TO POWER)
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (with the leading fence convention). For all $G_{tgt} \in [\![P_{tgt}]\!]_{\text{POWER}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21'}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Please see Appendix E for the proof.

Finally, we can prove the compilation correctness from JAM21 to Power via its observational equivalence relation with JAM21'.

▶ THEOREM 2 (COMPILATION CORRECTNESS TO POWER (LEADING FENCE))
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (using the leading fence convention). For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{POWER}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Please see Appendix E for the proof.

In addition, we prove that the trailing fence convention is also correct.

▶ COROLLARY 1 (COMPILATION CORRECTNESS TO POWER (TRAILING FENCE))
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (using the trailing fence convention). For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{POWER}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Please see Appendix E for the proof.

3.5 Compilation Correctness to x86-TSO

In this section we show that the current compilation scheme to x86-TSO is correct with respect to the TSO memory model.

3.5.1 The x86-TSO Model

We use the x86-TSO model defined in Herd7 [AMT14], and the full model can be found in Appendix F).

- po and rf are the same as defined previously
- co is the union of total orders among writes to the same location. Additionally, if i_1 and i_2 are events on different threads, then $i_1 \xrightarrow{\texttt{coe}} i_2$.
- $\texttt{fr} \triangleq \texttt{rf}^{-1}; \texttt{co}$ and $\texttt{com} \triangleq \texttt{rf}|\texttt{co}|\texttt{fr}$
- $i_1 \xrightarrow{\texttt{rmw}} i_2$ if and only if $i_1 \xrightarrow{\texttt{po}} i_2$ and i_1 and i_2 belongs to the same RMW event.
- $i_1 \xrightarrow{\texttt{po-ghb}} i_2$ if and only if $i_1 \xrightarrow{\texttt{po}} i_2 \wedge ((i_1 \in G.\mathsf{Wrts} \wedge i_2 \in G.\mathsf{Wrts}) \vee (i_1 \in G.\mathsf{Rds}))$.
- $i_1 \xrightarrow{\texttt{implied}} i_2$ if and only if $(i_1 \in G.\mathsf{Wrts} \wedge i_2 \in G.\mathsf{Rds} \wedge i_1 \xrightarrow{\texttt{po}} i_2) \wedge (i_1 \in G.\texttt{RMW} \vee i_2 \in G.\texttt{RMW})$
- $i_1 \xrightarrow{\texttt{mfence}} i_2$ if and only if $\exists i_3 . i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2 \wedge i_3$ is an `mfence`
- $\texttt{ghb} \triangleq \texttt{mfence} \mid \texttt{implied} \mid \texttt{po-ghb} \mid \texttt{rfe} \mid \texttt{fr} \mid \texttt{co}$

▶ Definition 8
An execution history G is TSO-*consistent* if it is trace coherent and satiesfies the following three requirements:

1. SC-per-location: `po-loc | com` is acyclic,
2. Atomicity: `rmw & (fre ; coe)` is empty,
3. Global Happens-Before: `ghb` is acyclic

`getOpaque()` $\rightsquigarrow$	`mov`
`setOpaque()` $\rightsquigarrow$	`mov`
`getAcquire()` $\rightsquigarrow$	`mov`
`setRelease()` $\rightsquigarrow$	`mov`
`getVolatile()` $\rightsquigarrow$	`mov`
`setVolatile()` $\rightsquigarrow$	`mov ; mfence`
`AcquireFence()` $\rightsquigarrow$	$NoOp$
`ReleaseFence()` $\rightsquigarrow$	$NoOp$
`fullFence()` $\rightsquigarrow$	`mfence`
`getAndAdd()` $\rightsquigarrow$	`lock xaddl`
`getAndAddAcquire()` $\rightsquigarrow$	`lock xaddl`
`getAndAddRelease()` $\rightsquigarrow$	`lock xaddl`

Figure 3.6: Compilation to x86-TSO

We say such execution history G is *allowed* by TSO. Otherwise, it is *forbidden*.

3.5.2 Compilation Scheme

We use the following compilation scheme. Note that instead of using `mfence` instruction for full fences, HotSpot uses a read-modify-write instruction to emulate the synchronization effect of it. According to the definition of TSO, the synchronization effect of a RMW event is exactly the same as an `mfence` event. Both of them produce a `ghb` order before and after the event. Therefore, we keep the simplicity of the proof here by using the `mfence` instruction.

3.5.3 Proof of Compilation Correctness

▶ Definition 9 (Compilation of an Execution)
Given a Java program P_{src}, let P_{tgt} be the target-level program compiled from P_{src} using the compilation scheme to x86 as shown above. Let G_{src} be a candidate execution of P_{src} and G_{tgt} be a candidate execution graph of P_{tgt}. We say $G_{src} \rightsquigarrow G_{tgt}$ if:

1. $G_{tgt}.\texttt{IW} = G_{src}.\texttt{IW}$ and $G_{tgt}.\texttt{FW} = G_{src}.\texttt{FW}$
2. $G_{tgt}.\textsf{Evts}\backslash G_{tgt}.\textsf{F} = G_{src}.\textsf{Evts}\backslash G_{src}.\textsf{F}$, $G_{tgt}.\texttt{rf} = G_{src}.\texttt{rf}$, and $G_{src}.\texttt{po} \subseteq G_{tgt}.\texttt{po}$

3. $G_{tgt}.\texttt{co} \subseteq G_{src}.\texttt{to}$

4. If $i_1, i_2 \in G_{src}$.Evts and $i_1 \xrightarrow{\text{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\text{po}} i_3 \xrightarrow{\text{po}} i_2$ for $i_1, i_2 \in G_{tgt}$.Evts and $i_3 \in G_{tgt}$.F where i_3 is an event stem from an `mfence` instruction.

Note that this definition does not say anything about whether an execution graph is consistent under a memory model.

Similar to the previous section, we start by showing compilation correctness from JAM21′ to x86-TSO.

▶ LEMMA 2
Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{TSO}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}'}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

The proof can be found in Appendix G.

▶ THEOREM 3
Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{TSO}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

The proof can be found in Appendix G.

3.6 Performance Implications

At the time of writing, the compiler bug [Shi21] has been reported but still not resolved. The main argument against fixing the bug by inserting the missing fence instruction is that it may slow down the performance significantly. In this section, we argue that this is not the case.

The reason we only translated our `volatile-non-sc` example to Power instructions is that we only expect changes in the implementation of compilers targeting Power architectures. There is no need to change the Java compilers for x86 [SSO10] and ARMv8 [PFD18] all thanks to a property called *write atomicity*. Write atomicity, or *multicopy atomicity*, ensures that, when a write issued by a thread becomes observable by any other thread, it is observable by all other threads in the system. The issue that we demonstrate in this paper is caused by a write operation becoming visible to some threads before some other threads. Therefore, this violation of sequential consistency may only be observed when compiling to non-multicopy atomic architectures. If the underlying architecture ensures multicopy atomicity, then we can be sure that all writes are committed in a broadcast style and Release-Acquire semantics is sufficient. Since x86 [SSO10] and ARMv8 [PFD18] are multicopy atomic, we do not expect the incorrect program behavior to appear on those architectures. Therefore, no change is needed in compilers targeting multicopy-atomic architectures. In fact, we give a correctness proof for x86 in Appendix 3.5 to concretely show that the current compilation scheme to x86 is correct with respect to the x86-TSO memory model. Furthermore, the fence instruction that compilers use to compile to ARMv7 is the `DMB SY` instruction [Lea11b], which captures the same effects of a `fullFence()`. The only change that needs to be made is when compiling to Power instructions. This change might slow down some programs. However, relative to all other major factors that affect the performance of Java programs, we expect the impact by this change in compilers to be small.

Furthermore, symmetric to "leading fence" scheme, the "trailing fence" scheme is also valid. A correct compiler may choose to either of the schemes. Usually one may wish to choose the "trailing fence" scheme for better performance. In this case, comparing to the original compilation scheme, the fix only changes the compilation scheme for each Volatile read:

1. Remove the `hwsync` in front of the `lwz` instruction
2. Change the `lwsync` following the `lwz` instruction to `hwsync`

It is easy to see that this fix only requires, in effect, moving the `hwsync` instructions that were originally inserted before the `lwz` instruction, but does not add more. In addition, it removes the `lwsync` instructions. Therefore, we do not expect this change to the compilation scheme to have much performance impact as argued in the discussions in the bug report [Shi21].

On the other hand, the impact of this change for compiler optimizations is unclear. That is, whether this revised compilation scheme disables some of the compiler optimizations is still a question. However, since C/C++11 compilers has long adopted this compilation scheme and performance has always been the first priority in their implementations, the possibility of disabling optimisations is unlikely. We leave a detailed empirical study for future work.

CHAPTER 4

Soundness of Predictive Analyses

4.1 Introduction

Concurrent programs are often error-prone due to the non-deterministic nature of their executions. Over the last two decades, various techniques have been proposed to catch bugs in concurrent programs. Among them, *dynamic predictive analysis* has become a promising research area. A dynamic predictive analysis takes an execution trace as input and discovers concurrency bugs without accessing the program source code. Recent work in the area of predictive analysis [MPV21, HMR14, HLR15, CYW21, Pav20, TMP23, GRX19, KMV17, SES12, MKV18, FF09] support *soundness* as one of their most important properties. A sound predictive analysis reports no false positives, which sounds straightforward but has been defined in many different ways in previous work [SES12, KMV17, Pav20, HMR14, SCR13, MKV18, MPV21, TMP23].

Three problems emerge from the use of varying soundness criteria in previous work. First, there is no single recipe for proving soundness, so existing proofs cannot be directly applied or adapted to a new algorithm. Second, some previous proofs later turned out to be flawed when counterexamples emerged, without revealing whether the algorithm or the soundness proof were wrong. Third, existing soundness criteria and proof techniques have no support for weak memory models. This is in part due the focus on trace-based soundness definitions in previous work, which is a poor fit for weak memory models.

To address these issues, we propose a modular framework for soundness of predictive analyses that unifies, simplifies, and generalizes existing approaches and handles weak memory behaviors.

Instead of traces, our framework is based on *execution graphs*, which are widely used in the field of axiomatic memory models [AMS12, ADG21, PFD18, PLV19, AFI09]. Compared to traces, execution graphs distinguish the sequential semantics per thread versus global weak memory semantics as two separate aspects of validity. The former is affected by the source language semantics, and the latter is determined by the axiomatic memory model. As a result, this feature enables a modular structure of soundness, which also leads to a simple recipe for constructing a proof.

Our contributions are the following:

§4.4 We propose a modular soundness definition parameterized by a memory model. The definition is more general than existing soundness definitions, reflects a closer connection with the language semantics, and can be applied with weak memory models. Specifically, we derive an executability property from a language semantics and augment existing multicopy-atomic memory models with lock semantics. We analyze the execution spaces represented by the existing soundness definitions and explain their relationships with our soundness definition.

§4.5 We provide a three-step recipe for constructing a soundness proof by constructing a witness execution graph that satisfies our soundness definition. We also provide a set of reusable lemmas that can be applied in the construction of proofs under other memory models or semantic constraints.

§4.6 We use our recipe to prove the soundness of six data race predictive analyses [Lam78, MKV18, MPV21, Pav20, HMR14, HH16]. Among them, the proof for MCR-TSO [HH16] is the first proof of soundness for a predictive analysis that works with weak memory. In addition, we extend the approach of [HH16] and define a new data race predictor based on a transformation that can discover more data races under TSO. We show that the extended data race predictor is sound following the same recipe in Appendix P.

Name	Memory Model	Bug Type	Sound Witness Definition	Ref.
HB	SC	Data Race	Relaxed CR	[Lam78, FF09]
CP	SC	Data Race	Correct Reordering	[SES12]
WCP	SC	Data Race	Correct Reordering	[KMV17]
SHB	SC	Data Race	Relaxed CR	[MKV18]
SYNCP	SC	Data Race	Sync-Preserving CR	[MPV21]
SPD	SC	Deadlock	Sync-Preserving CR	[TMP23]
M2	SC	Data Race	Correct Reordering	[Pav20]
RVPREDICT	SC	Data Race	Feasible Closure	[HMR14]
MCR-TSO	TSO	Data Race	Correct Reordering informally relaxed with TSO semantics	[HH16]

Figure 4.1: Various Criteria for a Sound Witness from Past Work

4.2 Motivation

A concurrency bug is a sequence of events that occur in some specific order. For example, a *data race* is a pair of two conflicting events ordered consecutively. Since a predictive analysis predicts whether a bug can occur in some execution of the program that produced the input execution, the soundness theorem of an analysis is defined by the existence of a *witness* execution that exhibits the bug. Therefore, characterizing the valid witness executions and showing that each reported bug corresponds to a witness execution that satisfies all the characteristics become critical in the soundness proofs of existing works in predictive analysis. Fig. 4.1 shows a summary of the soundness criteria used in existing works.

FASTTRACK [FF09] is a data race analysis that builds a partial order, the *happens-before* (HB) order [Lam78], among the events in a given input trace. While FASTTRACK with HB can report multiple data races in a single run, only the first data race is guaranteed to be sound. The soundness theorem of FASTTRACK states that the algorithm correctly implements the HB order using vector clocks. On the other hand, the soundness of the HB order was assumed in the paper.

After FastTrack, CP [SES12] and WCP [KMV17] built weaker partial orders than HB and used Correct Reordering (CR) to characterize valid witness traces. Correct Reordering requires each read event that appears in the witness execution to maintain the same values as in the input execution. The soundness theorems of both algorithms state that the first race reported by the algorithms is a HB-race or there is a deadlock in a correct reordering of the input trace. Both the soundness of Correct Reordering and the HB order were implicitly assumed.

[MKV18] shows an improvement over HB that ensures the soundness of *all* reported data races. Their idea is to build a strictly stronger partial order, *schedulable-happens-before* (SHB), which orders the unsound HB-races after the first race, so that all reported data races are sound. The soundness of SHB is based on a relaxed version of Correct Reordering, denoted as Relaxed CR in Fig. 4.1. Relaxed CR has the same requirements of Correct Reordering except that if a read is the last event of a thread in the witness execution, then it does not have to maintain the same value. In addition to the soundness of SHB, [MKV18] also formally proved that HB is sound under the definition of Relaxed CR. But the soundness of Relaxed CR and Correct Reordering were implicitly assumed.

[MPV21] and [TMP23] used a more restricted version of Correct Reordering called Sync-Preserving CR. In addition to the requirements of Correct Reordering, the critical sections that appear in a Sync-Preserving CR follow the same synchronization order as in the input trace. The proofs of both works were done by linearizing an event closure called SRFClosure (or SPClosure in [TMP23]) using the same trace order of the input trace. The soundness of Sync-Preserving CR relied on Correct Reordering, which was implicitly assumed.

[Pav20] used Correct Reordering as the soundness criteria for a witness trace. The algorithm computes a partial order P over a subset of events from the input trace and used it to determine if a pair of events forms a data race. The key correctness result is their Theorem 3.1, which showed that for a trace-closed partial order P computed from the input trace, the Max-Min algorithm that solves a particular linearization problem based on P always produce a correct reordering [Pav20, Theorem 3.1]. But the soundness of Correct Reordering was implicitly assumed.

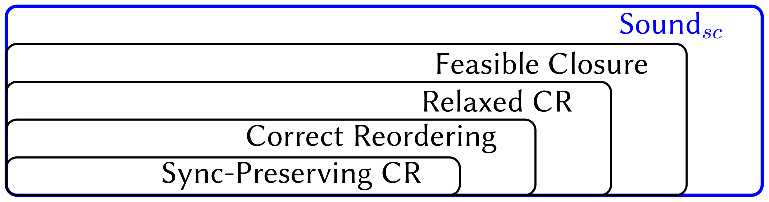


Figure 4.2: Hierarchy of Existing Sound Definitions under Sequential Consistency

[HMR14] used the notion of Feasible Closure of traces derived from the causal model of [SCR13]. Their main correctness result [HMR14, Theorem 1] showed the existence of a mapping from symbolic feasible traces to concrete feasible traces, but did not formally show that the set of concrete feasible traces can be generated by the same program of the input trace.

[HH16] used a restricted version of Feasible Closure that preserves all read values, but relaxed with a memory operation constraint, Φ_{mem}, to capture the weak behavior under x86-TSO. Effectively, their soundness criteria is the same as Correct Reordering relaxed with the TSO write-buffer semantics. Instead of requiring all program orders to be preserved, it does not require write-to-read program orders to be preserved. However, the soundness of the constraints was left unproved.

From the review above, we can see that the soundness criteria from existing works all specify properties that a witness execution has to satisfy given an input execution. For dynamic predictive analyses, the program source code is kept unknown. Therefore, all existing soundness definitions focus on characterizing a set of valid executions that can be produced by *all* programs that can generate the input execution. In other words, existing soundness definitions focus on executions in an *intersection* space of all the programs that can generate the input execution. Fig. 4.2 shows a hierarchy of the sets of valid executions specified by the soundness criteria from past works. Each rectangle represents a set of executions that satisfiy the corresponding soundness definition.

However, since all existing definitions focus on an intersection execution space, the size of this execution space depends on the amount of information about the program semantics captured in the input execution. Indeed, the analyses that used Feasible Closure [HMR14, SCR13, HLR15] record branch events in the input execution whereas other analyses [Pav20, MKV18, MPV21, TMP23, KMV17, SES12] do not. This difference enables Feasible Closure to subsume other soundness definitions in Fig. 4.2. A better approach is to define soundness independently of how much information the input execution captures. We focus on characterizing the set of valid executions that can be produced by a single program that generates the input execution. In other words, our soundness definition is *not based on an intersection* of execution sets, but is based on the complete execution set of each program that generates each input execution. Obviously, this set of executions subsumes all other sets specified by existing definitions, and is shown as the blue rectangle in Fig. 4.2 annotated with Sound_{sc}. The set inclusion relations shown in Fig. 4.2 is the following.

$$\begin{aligned}\text{Sound}_{sc}\ \text{[this paper]} &\supseteq \text{Feasible Closure [HMR14]}\\ &\supseteq \text{Relaxed CR [MKV18]}\\ &\supseteq \text{Correct Reordering [Pav20, KMV17, SES12]}\\ &\supseteq \text{Sync-Preserving CR [MPV21, TMP23]}\end{aligned}$$

The subscript sc in the name of our soundness stands for sequential consistency. While sequential consistency is subsumed by other memory models, the sets of valid executions under different memory models are not comparable in general. In §4.4, we give a general definition of soundness, $\text{Sound}_{\mathcal{M}}$, under a memory model $\mathcal{M}$.

The dynamic nature of the analyses, that is, having no access to the program source code, makes this set of executions unknown to the analysis tools. As a result, the traditional approach, which requires one to step through the source code of the program to show an execution is valid, cannot be used to prove a predictive analysis sound. Instead, we leverage the existing input execution and provide a three-step proof technique in Section 4.5. Note that the restriction of having no access to the program source code does not affect the soundness definition, but only the proof technique.

In Appendix L, we provide the formal definitions of the existing soundness criteria and show their relationships with our soundness definition.

4.3 Preliminaries

In this section, we recall standard definitions of programs and execution graphs [PLV19], which we extend with acquire and release events [Pav20], and a straightforward `sync` order over such events. Our operational semantics of programs uses the standard notion of a thread state (Definition 12). We instrument thread states with symbolic book-keeping information that plays no role in the semantics but helps in proofs of soundness.

We use standard notation for relations and functions. For a relation R, we use $R^?$, R^+, and R^* to denote the reflexive, transitive, and reflexive-transitive closures of R, respectively. We use R^{-1} for the inverse of R. Given a set A, $[A]$ is the identity relation on A, $\wp(A)$ is the power set of A. The composition of two relations R_1 and R_2 is written as $R_1; R_2$. $R|_A$ stands for R restricted to the set A. We say a set A is downward-closed with respect to a relation R if for each element $e \in A$, if there is $\langle e', e \rangle \in R$, then $e' \in A$. We use the following domains: Loc is a set of shared memory locations; Lock is a set of locks; Val is a set of concrete integer values; Sym is a set of symbols; Reg is a set of thread-local registers; and Thrd is a set of natural numbers for thread identifiers. In addition, we assume all memory locations are fixed.

4.3.1 Programs

A concurrent program consists of a set of threads, each containing a list of instructions. Formally, a *program* is a map from thread identifiers to sequential programs P : Tid $\rightarrow$ Sprog, where Sprog $= \mathbb{N} \rightarrow$ Instr and the set of instructions is defined in Fig. 4.3.

Our sequential programs use standard instructions, including `lock` and `unlock` instructions for the acquire and release operations of locks. These two instructions allow us to reason about lock operations without worrying about the implementations of locks. We assume the imple-

$$
\begin{array}{lll}
i \in \mathsf{Instr} & ::= & r \texttt{ := } e \\
& \mid & \texttt{if } e \texttt{ goto } n \\
& \mid & [x] \texttt{ := } e \\
& \mid & r \texttt{ := } [x] \\
& \mid & \texttt{lock}(l) \mid \texttt{unlock}(l) \\
e \in \mathsf{Expr} & ::= & r \mid e + e \mid e - e \mid e * e \mid v
\end{array}
\qquad
\begin{array}{r}
x \in \mathsf{Loc} \\
v \in \mathsf{Val} \\
r \in \mathsf{Reg} \\
l \in \mathsf{Lock} \\
n \in \mathbb{N}
\end{array}
$$

Figure 4.3: Instructions of the Language

mentation of `lock` and `unlock` is correct and guarantees lock-fairness [LNO20, LCK23] so that each lock-acquiring request is eventually fulfilled.

4.3.2 Execution Graphs

Each program generates a set of *execution graphs*. In this section, we formally define events and execution graphs.

An *event* is a tuple of form $\langle tid, eid, typ, val, loc\rangle$ where $tid \in \mathbb{N}$ is the identifier of the thread of the event; $eid \in \mathbb{N}$ is a unique identifier for the event; $typ \in \{r, w, \mathsf{acq}, \mathsf{rel}, \mathsf{br}\}$ is the event type with r standing for read events, w for write events, acq for lock acquire events, rel for lock release events, and br for branch events; $val \in \mathsf{Val}$ is the value of the event; and $loc \in \mathsf{Loc} \cup \mathsf{Lock}$ is the memory location or lock that the event accesses.

An execution graph consists of a set of events and the relations over the events.

▶ Definition 10 (Execution Graph)
An *execution graph* $G = \langle \mathsf{Evts}, \mathsf{po}, \texttt{rf}, \texttt{co}, \texttt{sync}\rangle$ with each component defined below:

- Events is a finite set of events $G.\mathsf{Evts}$. We use $G.T$ where $T \in \{\mathsf{Rds}, \mathsf{Wrts}, \mathsf{Acqs}, \mathsf{Rels}, \mathsf{Brs}\}$ to denote subsets of events based on their types. In addition, $G.\mathsf{Init}$ is a set of initialization writes to each memory location. $G.\mathsf{Init} \cap G.\mathsf{Evts} = \varnothing$.
- Program Order (po) is a partial order $G.\mathsf{po} \subseteq G.\mathsf{Evts} \times G.\mathsf{Evts}$. $\langle e_1, e_2\rangle \in G.\mathsf{po}$ iff $e_1.tid = e_2.tid$ and $e_1.eid < e_2.eid$.

- Reads-from Order ($\texttt{rf}$) is a binary relation $G.\texttt{rf} \subseteq G.\mathsf{Wrts} \times G.\mathsf{Rds}$.
- Coherence Order ($\texttt{co}$) is a binary relation $G.\texttt{co} \subseteq G.\mathsf{Wrts} \times G.\mathsf{Wrts}$.
- Synchronization Order ($\texttt{sync}$) is a binary relation $G.\texttt{sync} \subseteq G.\mathsf{Rels} \times G.\mathsf{Acqs}$.

In addition, the from-read order [AMS12] ($\texttt{fr}$) is defined as $\texttt{fr} = \texttt{rf}^{-1}; \texttt{co}$. We use $\texttt{com}$ to denote the union of the communication orders, $\texttt{com} = \texttt{fr} \cup \texttt{rf} \cup \texttt{co}$. If an execution graph is sequentially consistent, then there is also a linear order $\texttt{trace}$ among all events of the execution graph. For each event e, we use $\mathsf{LocksHeld}(e)$ to denote the set of locks that are acquired but not released at the point of e in its thread.

Furthermore, a *symbolic execution graph* $\hat{G}$ is an execution graph with an event set of which event values are either concrete values $v \in \mathsf{Val}$ or symbols $\hat{s} \in \mathsf{Sym}$. For each read event $r \in \hat{G}.\mathsf{Rds}$, if r is a read event with a concrete value, then either there exists a unique concrete write event $\langle w, r\rangle \in \hat{G}.\texttt{rf}$ such that $w.loc = r.loc$ and $w.val = r.val$, or $r.val$ is the initial value. Note that for a read event with symbolic value, we do not require it to be mapped to a unique write event. Intuitively, symbolic execution graphs are used to accomodate the approaches of [HMR14] and [HH16], where some of the events becomes symbolic due to changes of the $\texttt{rf}$-map in the prediction result. On the other hand, later in Section 4.5.3, we show that there exists a concrete execution graph that a symbolic execution graph can be mapped to, given a set of conditions are satisfied.

Using definitions from [PLV19], an execution graph G is a *plain* execution graph, if $G.\texttt{rf} = G.\texttt{co} = G.\texttt{sync} = \varnothing$. An execution graph G with well-formed $\texttt{rf}$, $\texttt{co}$, and $\texttt{sync}$ relations is called a *complete* execution graph.

▶ Definition 11 (Well-formed Complete Execution Graph)
A complete execution graph $G = (\mathsf{Evts}, \mathsf{po}, \texttt{rf}, \texttt{co}, \texttt{sync})$ is well-formed if

- Each read event must be justified by a write event in the same execution graph. That is, for each read event $r \in G.\mathsf{Rds}$ either there is a unique write event $w \in G.\mathsf{Wrts}$ such that $\langle w, r\rangle \in G.\texttt{rf}$, or $r.val = w_{init}.val$ where w_{init} is the initialization write of $r.loc$. For each pair $\langle w, r\rangle \in G.\texttt{rf}$, $r.val = w.val$, and $r.loc = w.loc$.

- For each pair of distinct writes w_1 and w_2, if $w_1.loc = w_2.loc$, then either $\langle w_1, w_2\rangle \in G.\texttt{co}$ or $\langle w_2, w_1\rangle \in G.\texttt{co}$ but not both. In addition, $\langle w_0, w\rangle \in G.\texttt{co}$ for each initial write w_0 and $w \in G.\mathsf{Wrts}$ such that $w_0.loc = w.loc$.
- There is a function $\mathsf{match} : G.\mathsf{Rels} \to G.\mathsf{Acqs}$ such that for each $\mathsf{rel} \in G.\mathsf{Rels}$, $\mathsf{match}(\mathsf{rel}).loc = \mathsf{rel}.loc$, and $\langle \mathsf{match}(\mathsf{rel}), \mathsf{rel}\rangle \in G.\mathsf{po}$. There is a function $Open : G.\mathsf{Locks} \to G.\mathsf{Acqs}$ such that $Open(l) = \mathsf{acq}$ iff $\mathsf{acq}.loc = l$ and for all $\mathsf{rel} \in G.\mathsf{Rels}$, $\mathsf{match}(\mathsf{rel}) \neq \mathsf{acq}$. If no such acquire event exists for lock l, $Open(l) = \bot$. For each $\langle \mathsf{rel}, \mathsf{acq}\rangle \in G.\texttt{sync}$, $\mathsf{rel}.loc = \mathsf{acq}.loc$. For each lock $l \in \mathsf{Lock}$, let $\mathsf{cs} \in linear(G.\mathsf{Rels}_l)$ be a linear order among all the release events of l. Then for each $\langle \mathsf{rel}_1(l), \mathsf{rel}_2(l)\rangle \in \mathsf{cs}$, we have $\langle \mathsf{rel}_1(l), \mathsf{match}(\mathsf{rel}_2(l))\rangle \in \texttt{sync}$.

4.3.3 From Programs to Execution Graphs

Given the formal definitions of programs and execution graphs, we explain how execution graphs are generated from programs.

An execution graph is generated from a program by starting with an empty execution graph G_0. Given a value map $loadVal : Load \to \mathsf{Val}$ for each load instruction of the form $r := [x]$, use the sequential operational semantics of instructions and the $loadVal()$ function to generate a chain of events for each thread by adding one event at a time. At this stage, only the program order po is added to the graph while other relations are left empty. The result of this stage is a plain execution graph. Note that we do not consider the consistency of the graph at this stage. Lastly, `rf`, `co`, and `sync` relations are added to the plain execution graph according to the values and the well-formedness conditions, which we will define later. The result of this stage is a complete execution graph ready for the consistency check.

We next define the notion of thread state. Here we use the same definition from [PLV19] with an additional *symbolic* register state Φ^θ and a function $\theta : G.\mathsf{Rds} \to \mathsf{Sym}$. These two new components do *not* replace any functionality of other components as described in [PLV19]. They are only added for additional book-keeping purposes. Later in the proof of Lemma 3 in §4.4.3, they are used to re-construct state transition paths of prediction results.

In addition, we assume memory fairness [LNO20] so that each `lock`(l) instruction will eventually succeed and produce an acq event.

$$\hat{e} \in \mathsf{SymExpr} ::= \hat{s} \mid (\hat{e}) \mid v \mid \hat{e} + \hat{e} \mid \hat{e} - \hat{e} \mid \hat{e} * \hat{e} \qquad \hat{s} \in \mathsf{Sym} \quad v \in \mathsf{Val}$$

Figure 4.4: Symbolic Expressions

▶ Definition 12 (Thread State)
A thread state $st \in \mathsf{State}$ for a thread t is a tuple

$$st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^{\theta} \rangle$$

with each component defined as follows:

- $sprog : \mathbb{N} \rightarrow \mathsf{Instr}$ is the instructions in thread t, $sprog = P(t)$.
- $pc \in \mathbb{N}$ is the program counter pointing to the next instruction
- $\Phi : \mathsf{Reg} \rightarrow \mathsf{Val}$ is map recording the value of each register
- G is the execution graph that has been constructed so far for thread t
- $\Psi : \mathsf{Reg} \rightarrow \wp(G.\mathsf{Rds})$ maps each register to a set of read events in G such that the value in the register depends on the set of reads.
- $ctrl \subseteq G.\mathsf{Rds}$ is a set of read events that has a control dependency with the current program point.
- $\theta : G.\mathsf{Rds} \rightarrow \mathsf{Sym}$ is a map recording the symbolic value of each read event. When a read event is generated from executing a load instruction, in addition to the concrete value that it receives via the $loadVal$ function, a fresh new symbol is assigned to the read event and an entry is added in θ.
- $\Phi^{\theta} : \mathsf{Reg} \rightarrow \mathsf{SymExpr}$ is a map recording the symbolic expression used to calculate the current (concrete) value on each register. The grammar of symbolic expressions is defined in Fig. 4.4. In other words, the concrete value on each register can be calculated by plugging the concrete values of the read events into its symbolic expression.

The initial state for a thread $t \in \mathsf{Tid}$ is $st_0^t = \langle sprog, 0, \lambda r.0, G_0, \lambda r.\varnothing, \varnothing, \lambda rd.null, \lambda r.0 \rangle$ where $sprog = P(t)$ and G_0 is an empty execution graph such that $G_0.\mathsf{Evts} = G_0.\mathsf{po} = G_0.\mathtt{rf} = G_0.\mathtt{co} = G_0.\mathtt{sync} = \varnothing$.

An important invariant for a state to be valid is $\forall r \in \mathsf{Reg},\ subst(\Phi^{\theta}(r), val^{\theta}(\Psi(r))) = \Phi(r)$ where the helper functions $subst : \mathsf{SymExpr} \to (\mathsf{Sym} \to \mathsf{Val}) \to \mathsf{Expr}$ replaces each symbol in an symbolic expression with a concrete value given a mapping from symbols to concrete values, and $val^{\theta} : \wp(G.\mathsf{Rds}) \to (\mathsf{Sym} \to \mathsf{Val})$ uses θ to provide a set of such mapping. In words, evaluating the symbolic expressions that each register is mapped to with the concrete value of each read events should have the same result as the value stored in Φ. Note that in order for this invariant to hold, a register cannot occur on both sides of an assignment.

For two thread states $st_1, st_2 \in \mathsf{State}$ of a thread t, we write $st_1 \to_t st_2$ if st_1 steps to st_2 in a single step and $st_1 \to_t^* st_2$ if st_1 steps to st_2 in zero or more steps. We provide the sequential operational semantics for our language in Appendix M.

We borrow the following definition from [PLV19]. For an execution graph G and $t \in \mathsf{Tid}$, $G|_t$ is a thread of events such that $G|_t.\mathsf{Evts} = G.\mathsf{Evts}|_t$, $G|_t.\mathsf{po} = G.\mathsf{po} \cap (G|_t.\mathsf{Evts} \times G|_t.\mathsf{Evts})$, and $G|_t.\mathtt{rf} = G|_t.\mathtt{co} = G|_t.\mathtt{sync} = \varnothing$

▶ Definition 13 (Plain Program Executions [PLV19])
An execution graph G is an *execution graph of a program* P if for every $t \in \mathsf{Tid}$, there exists a state st such that $st.G = G|_t$ and $st_0^t \to_t^* st$.

We write $G \in [\![P]\!]$ for such an execution graph G and program P. We say a plain symbolic graph $\hat{G} \in [\![P]\!]$ if there exists a $st_0 \to_t^* st_n$ transition path for each thread t that produces the resulting graph. The only difference from Definition 13 is the invariant condition that each state has to maintain because $val^{\theta}(\Psi(r))$ may not be defined for every register in the presence of events with symbolic values. We relax the requirement and let the invariant condition only apply to registers with concrete values, i.e., $subst(\Phi^{\theta}(reg), val^{\theta}(\Psi(reg))) = \Phi(reg)$ if $\Phi(reg) \in \mathsf{Val}$. Moreover, if a write event has a concrete value, it has to be computed from concrete values as well. In other words, for each state st_i such that $st_i.sprog(st_i.pc) = [x]\ := e$, if w is the write event emitted after st_i and $w.val \in \mathsf{Val}$, then for each $reg \in \mathsf{Reg}$ used in e, $\Phi(reg) \in \mathsf{Val}$.

Given a plain execution graph, the `rf`, `co`, and `sync` relations are added to the graph according to Definition 11 to obtain a complete execution graph.

4.3.4 Bug Sequence

Each instance of bug is represented as a sequence of events:

$$b = e_1 \dots e_n$$

The basic well-formedness requirement for a bug sequence is for it to be sequentially consistent. We give three common examples below.

Data Races. In the context of *data race* prediction, each reported bug is in the form of a sequence $e_1 e_2$ such that $e_1.tid \neq e_2.tid \wedge e_1.loc = e_2.loc \wedge \{e_1.typ, e_2.typ\} \cap \{\mathsf{Wrts}\} \neq \varnothing$. We write $e_1 \bowtie e_2$ for such pair of events. Note that the value of e_1 or e_2 may not be the same as occurred in the input execution G_σ.

Deadlocks. In the context of *deadlock* prediction, [TMP23] provided a necessary pattern consisting a set of acquire events that imposes a cyclic resource dependency. However, note that this pattern itself violates memory consistency because a lock cannot be acquired again while it is held. The essential issue here is that, the acquire events in a sound execution represent successful aquisition of the locks, whereas the acquire events in the deadlock pattern of [TMP23] represent *requests* of the locks. Therefore, for deadlock prediction, the lock request events have to be distinguished from lock acquire events, following the approach of [KP18]. The rest of the definition stays the same as in [TMP23]. A bug sequence of size k is defined as a sequence of request events $e_0, \dots, e_{k-1}$ on k distinct threads $t_0, \dots, t_{k-1}$ and k distinct locks $l_0, \dots, l_{k-1}$ such that $e_i.tid = t_i$, e_i is a request of lock l_i, where $l_i \in \mathsf{LocksHeld}(e_{(i+1)\%k})$. In addition, $\mathsf{LocksHeld}(e_i) \cap \mathsf{LocksHeld}(e_j) = \varnothing$ for $i \neq j$. Note that using this definition, the composition of a sound witness execution and the deadlock sequence is still a sound execution.

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[d] \\
G_\sigma \arrow[r, "\text{Predict}"'] & G_\rho
\end{tikzcd}$$

Figure 4.5: Predictive Analyses

Atomicity Violations. Atomicity violation patterns can be represented as sequences of events in multiple ways. Each pattern consists of an atomic pair, i.e., a pair of events (e_1, e_2) on the same thread that is expected to be executed atomically, and a third event e_3 accessing the same memory location from another thread. [HLR15] provided one pattern in their example: $r_1 w_1 w_2$ where $r_1 \in \mathsf{Rds}$, $w_1, w_2 \in \mathsf{Wrts}$, $r_1.loc = w_1.loc = w_2.loc$, and $\langle r_1, w_2 \rangle$ is an atomic pair. [CYW21] provided another example: $w_2 w_1 r_1$ where $\langle w_2, r_1 \rangle$ is an atomic pair.

4.4 Soundness

We begin this section by asking a question.

What does it mean for a predictive analysis to be sound?

A sound predictive analysis only reports a bug if it can be exposed by a valid witness execution. As shown in Fig. 4.5, let P be a program with a reported bug to be fixed. G_σ is an execution graph of P captured from running P. In the rest of this paper, we assume the captured G_σ is sequentially consistent and the events of G_σ all have concrete values. A predictive analysis algorithm analyzes G_σ without inspecting P to spot the existence of any concurrent bug. Note that the predictive analysis can report bugs that are not necessarily exposed in the recorded execution G_σ, but in some other execution of the same program P. Let G_ρ be such a *witness* execution where the bug is exposed. Then a sound predictive analysis should ensure that G_ρ is indeed a valid execution of P. But what does it mean for an execution to be valid?

We identify two important but separate aspects in answering this question:

- **Executability** The program that generates the input execution has to be able to generate the witness execution. From Definition 13, we can see that this is a *local* property. That is, an execution can be generated by a program if each of its threads can be generated by the program. In an execution graph, threads are formed by events related by the program orders po. In other words, executability means the events on each thread can be generated by the program in the order specified by po.
- **Memory Consistency** Another factor is the memory consistency model under which the execution is executed. Axiomatic memory models specify consistent executions under weak memory contexts by characterizing the ordering relations among the events, including the inter-thread communications. Therefore, this is a *global* property that concerns the orders among the events in an execution graph.

In addition, the well-formedness of the witness and its relationship with the reported bug are required to ensure that the composition of the witness with the reported bug sequence is a sound execution. As a result, we define the overall soundness of a predictive analysis as a conjunction of four major components.

Formally, a predictive analysis is a function $\mathsf{Predict} : \mathcal{G} \rightarrow \wp(\mathcal{B})$ that takes a recorded execution as an input and reports a set of bugs predicted from the execution. Soundness of such a predictive analyses is defined as the following.

$$\begin{aligned}\mathsf{Sound}_{\mathcal{M}} \triangleq\ & \forall P \in \mathsf{Prog},\ \forall G_\sigma \in [\![P]\!] \wedge \text{SC-consistent}(G_\sigma),\ \forall b \in \mathsf{Predict}(G_\sigma),\\ & \exists G_\rho,\ G_\rho \rhd b \wedge \mathsf{wf}(G_\rho)\ \wedge\ G_\rho \in [\![P]\!]\ \wedge\ \mathcal{M}\text{-consistent}(G_\rho)\end{aligned}$$

The above soundness definition states that in order to prove a predictive analysis is sound, one needs to show that for each bug reported, there is a *witness* execution G_ρ such that:

§4.4.1	$G_\rho \rhd b$	G_ρ is composable with the reported bug b	[Definition 15]
§4.4.2	$\mathsf{wf}(G_\rho)$	G_ρ is well-formed	[Definition 16]
§4.4.3	$G_\rho \in [\![P]\!]$	G_ρ is executable	[Definition 13]
§4.4.4	$\mathcal{M}$-consistent(G_ρ)	G_ρ is $\mathcal{M}$-consistent	[Definition 18]

The first three parts of the conjunction states properties that the witness execution has to satisfy as a *plain* execution graph, whereas the last part is about the orders among the events in the witness execution as a *complete* execution graph. In the rest of this section, we discuss each part in details. The proofs of the propositions and lemmas can be found in Appendix N.

4.4.1 Composability with bug sequences

Intuitively, witness G_ρ is the execution that must occur before the bug sequence b occurs. Hence, the composition of them should also be a valid execution. We start by formally defining the composition of an execution graph with an event sequence.

▶ Definition 14 (Composition)
Let G_ρ be a well-formed execution graph and b be a bug sequence. Then the composition

$$G = G_\rho \circ b$$

is an execution graph such that:

- $G.\mathsf{Evts} = G_\rho.\mathsf{Evts} \cup b.\mathsf{Evts}$
- $G.\mathsf{po} = G_\rho.\mathsf{po} \cup b.\mathsf{po} \cup \{\langle e_1, e_2\rangle \mid e_1 \in G_\rho.\mathsf{Evts} \land e_2 \in b.\mathsf{Evts} \land e_1.tid = e_2.tid\}$
- $G.\mathtt{co} = G_\rho.\mathtt{co} \cup b.\mathtt{co} \cup \{\langle w_1, w_2\rangle \mid w_1 \in G_\rho.\mathsf{Wrts} \land w_2 \in b.\mathsf{Wrts} \land w_1.loc = w_2.loc\}$
- $G.\mathtt{rf} = G_\rho.\mathtt{rf} \cup b.\mathtt{rf} \cup \{\langle w, r\rangle \mid r \in b.\mathsf{Rds} \land r \notin range(b.\mathtt{rf}) \land w \in G_\rho.\mathsf{Wrts} \land (\forall w' \in G_\rho.\mathsf{Wrts}, (w'.loc = w.loc \land w' \neq w) \Rightarrow \langle w', w\rangle \in G_\rho.\mathtt{co}) \land w.loc = r.loc \land w.val = r.val\}$
- $G.\mathtt{sync} = G_\rho.\mathtt{sync} \cup b.\mathtt{sync} \cup \{\langle \mathsf{rel}(l), \mathsf{acq}(l)\rangle \mid \mathsf{rel}(l) \in G_\rho.\mathsf{Rels} \land \mathsf{acq}(l) \in b.\mathsf{Acqs}\}$

To ensure that the composition $G_\rho \circ b$ is executable, we require G_ρ to be *composable* with the bug sequence b.

▶ Definition 15 (Composability)
Let G_σ be an input execution that is sequentially consistent, G_ρ be a witness execution with an event map $\delta_\rho : G_\rho.\mathsf{Evts} \rightarrow G_\sigma.\mathsf{Evts}$, and b be a well-formed bug sequence with an event map $\delta_b : b.\mathsf{Evts} \rightarrow G_\sigma.\mathsf{Evts}$.

We say an execution graph G_ρ witnesses a bug sequence b, written $G_\rho \rhd b$, if $G_\rho.\mathsf{Evts} \cap b.\mathsf{Evts} = \emptyset$ and

- No Skipping. For each thread $t \in b.\mathsf{Thrd}$, let $e \in b|_t.\mathsf{Evts}$ be the first event occur in $b|_t$. For any $e' \in G_\sigma.\mathsf{Evts}$ such that $\langle e', \delta_b(e)\rangle \in G_\sigma.\mathsf{po}$, there is an event $e'' \in G_\rho.\mathsf{Evts}$ such that $\delta_\rho(e'') = e'$, and
- Same Control Flow. For each $r \in G_\rho.\mathsf{Rds}$ and $e \in b.\mathsf{Evts}$, if $\langle \delta_\rho(r), \delta_b(e)\rangle \in G_\sigma.\mathtt{ctrl}$, then $\delta_\rho(r) = r$.

The conditions above ensure the composition $G_\rho \circ b$ inherits executability from G_ρ.

▶ Proposition 1
Let G_ρ be a well-formed execution graph such that $G_\rho \in [\![P]\!]$, and b be a bug sequence. If $G_\rho \rhd b$, then $(G_\rho \circ b) \in [\![P]\!]$.

On the other hand, $\mathcal{M}$-consistency (which will be defined in §4.4.4) is inherited by construction.

▶ Proposition 2
Let G_ρ be a well-formed execution graph such that G_ρ is $\mathcal{M}$-consistent, and b be a sequence of events. Then $(G_\rho \circ b)$ is $\mathcal{M}$-consistent.

4.4.2 Well-formedness of Plain Execution

The second requirement is well-formedness requirements for a plain execution $\langle \mathsf{Evts}, \mathsf{po}\rangle$.

▶ Definition 16 (Well-Formed Plain Execution)
A plain execution $G_\rho = (\mathsf{Evts}, \mathsf{po})$ is well-formed if:

- $G_\rho.\mathsf{po}$ is a partial order over Evts that orders each pair of events $\langle e_1, e_2\rangle$ iff $e_1.tid = e_2.tid$.

- Read Feasible. For each read event included in G_ρ.Evts, either there exists a write event included in G_ρ.Evts with the same value and location, or the value of the read is the same as the initial value of the memory location.
- Lock Feasible. For each lock $l \in$ Locks, there is at most one open critical section protected by l. An open critical section is defined as sequence of events totally ordered by G_ρ.po where the minimal event in the sequence is an acquire event acq(l) for some lock l and the matching release event rel(l) is not included in G_ρ.Evts.

4.4.3 Executability

Now we turn our attention to the most important part of the soundness definition, executability. Given a witness execution graph G_ρ, we want to make sure that G_ρ is indeed an execution graph of the input program P, namely, $G_\rho \in [\![P]\!]$, via the semantics defined in Appendix M.

Typically, in order to show that an execution graph G_ρ is generated by a program P, one has to start from the initial state (as seen in §4.3) and determine whether there is a reachable state containing G_ρ from the initial state via a path of transitions. However, in the setting of dynamic analysis, one cannot inspect the source code of P and hence cannot follow the semantic rules to determine whether such a state is reachable.

In the setting of dynamic analysis, what we have is the input execution G_σ, which is obtained by running the program P, i.e., $G_\sigma \in [\![P]\!]$. Therefore, for each thread t in G_σ, there exists a path $st_0 \to st_1 \to \ldots \to st_k$ with each $st_i \in$ State and $st_0.sprog = P(t)$. We use $Path(G_\sigma|_t)$ to denote this path. Again, due to the nature of dynamic analyses, the states on this path are opaque. To show that $G_\rho \in [\![P]\!]$, the key is to *reuse* these states and identify a *similar* transition path that generates each thread of G_ρ from the initial state. By Definition 13, it means we have to show that for each thread t in G_ρ, there exists a state st'_m such that $st'_m.G = G_\rho|_t$ and $st'_0 \to_t^* st'_m$ with $st'_0.sprog = P(t)$. Note that st'_m is not necessarily a terminal state.

We first borrow the notion of *data-abstract equivalence* from [HMR14] and lift it to execution graphs.

$$\begin{tikzcd}
st_0 \arrow[r] \arrow[d, phantom, "\parallel"] & st_1 \arrow[r] \arrow[d, phantom, "\wr"] & \ldots \arrow[r] & st_m \arrow[r] \arrow[d, phantom, "\wr"] & \ldots \arrow[r] & st_k \\
st'_0 \arrow[r] & st'_1 \arrow[r] & \ldots \arrow[r] & st'_m & &
\end{tikzcd}$$

Figure 4.6: Similar State Transitions

▶ Definition 17 (Data-Abstract Equivalent Graph)
A plain execution graph G is *data-abstract equivalent* to another plain execution graph G' if there is a map $\delta : G.\mathsf{Evts} \rightarrow G'.\mathsf{Evts}$ such that for each event $e \in G.\mathsf{Evts} = \langle tid, eid, typ, val, loc\rangle$, there is $\delta(e) \in G'.\mathsf{Evts} = \langle tid, eid, typ, val', loc\rangle$ for some val'. In addition, for each $\langle e_1, e_2\rangle \in G.\mathsf{po}$, $\langle \delta(e_1), \delta(e_2)\rangle \in G'.\mathsf{po}$ and vice versa.

We write $G \approx G'$ if G is data-abstract equivalent to G.

The data-abstract equivalence relation can be extended naturally to the map Ψ and the set $ctrl$ in a state. Now we use this notion to define a similarity relation over states. Let $st, st' \in \mathsf{State}$ where $st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^{\theta}\rangle$ and $st' = \langle sprog', pc', \Phi', G', \Psi', ctrl', \theta', \Phi^{\theta\prime}\rangle$. We say $st' \approx st$, if $sprog = sprog'$, $pc = pc'$, $G \approx G'$, $\Psi \approx \Psi'$, $ctrl \approx ctrl'$, $\theta \approx \theta'$, and $\Phi^{\theta} = \Phi^{\theta\prime}$. For each read event $r \in G.\mathsf{Rds}$, $\theta(r) = \theta(\delta(r))$. Essentially, two states are data-abstract equivalent if they only differ by the concrete values on each read events. It's easy to see that if two states are identical, then they are data-abstract equivalent, i.e., $st = st' \Rightarrow st \approx st'$.

As illustrated in Fig. 4.6, for each thread t, if we can construct a similar state transition path $st'_0 \rightarrow \ldots \rightarrow st'_n$ that generates $G_{\rho}|_t$ from an empty state containing the same source program as $P(t)$, then by Definition 13 we can conclude that $G_{\rho} \in [\![P]\!]$.

We can use the following lemma to prove that an execution graph is executable.

▶ Lemma 3
Let G_{ρ}, G_{σ} be well-formed execution graphs and $G_{\sigma} \in [\![P]\!]$. If for each thread $t \in G_{\rho}.\mathsf{Thrd}$, $t \in G_{\sigma}.\mathsf{Thrd}$ and there is a $\sqsubseteq$-ordered set $\{st'_0, \ldots, st'_m\}$ such that for $i \in 0 \ldots m$,

- each state st'_i satisfies the invariant $subst(\Phi^{\theta}(r), val^{\theta}(\Psi(r))) = \Phi(r)$ for $r \in \mathsf{Reg}$,

- for each st'_i there exists a state $st_i \in Path(G_\sigma|_t)$ with $st_i \approx st'_i$ and $st_0 = st'_0$,
- for each st'_i if $st'_i.sprog(st'_i.pc) =$ `if` $expr$ `goto` k then $st_i.\Phi(expr) = st'_i.\Phi(expr)$,
- $st'_m.G = G_\rho|_t$,

then $G_\rho \in [\![P]\!]$.

Lemma 3 uses the notion of *states* to reason about the relation between the input execution and the witness execution. In addition, it requires each event in the witness graph to be concrete. The following lemma simplifies the requirements of executability, focusing the reasoning process on *events* and allowing events with symbolic values.

▶ Lemma 4
Let $\hat{G}_\rho$ be a well-formed symbolic execution graph and G_σ be a concrete input execution graph such that $G_\sigma \in [\![P]\!]$. If $\hat{G}_\rho$ satisfies the following conditions:

- there is a map $\delta : \hat{G}_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ such that for each event $e \in \hat{G}_\rho.\mathsf{Evts}$, $\delta(e) \approx e$ and if $e.val \in \mathsf{Val}$ (i.e., $e.val$ is concrete), then $\delta(e) = e$.
- if $\langle e_1, e_2 \rangle \in G_\sigma.\mathsf{po}$ and $e_2 = \delta(e'_2)$ for some $e'_2 \in \hat{G}_\rho.\mathsf{Evts}$, then there is an event $e'_1 \in \hat{G}_\rho.\mathsf{Evts}$ such that $e_1 = \delta(e'_1)$ and $\langle e'_1, e'_2 \rangle \in \hat{G}_\rho.\mathsf{po}$,
- for each thread $t \in \hat{G}_\rho.\mathsf{Thrd}$ and each event $e \in \hat{G}_\rho|_t.\mathsf{Evts}$, if there is a read event $r \in \hat{G}_\rho|_t.\mathsf{Rds}$, such that $\langle \delta(r), \delta(e) \rangle \in G_\sigma.\mathtt{ctrl}$, then $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete),
- for each write event $w \in \hat{G}_\rho.\mathsf{Wrts}$, if $w.val \in \mathsf{Val}$ (i.e., $w.val$ is concrete), then for all $r \in \hat{G}_\rho.\mathsf{Rds}$ such that $\langle \delta(r), \delta(w) \rangle \in G_\sigma.\mathtt{data}$, $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete).

then $\hat{G}_\rho \in [\![P]\!]$.

4.4.4 Memory Consistency

The second major component of soundness is memory consistency. Most existing work in predictive analysis focuses on sequentially consistent executions (except for [HH16], which focused on TSO and PSO models). Under sequential consistency, an execution can be treated as a linear sequence of events, i.e., a *trace*. However, under relaxed memory models, executions may not be linearizable by a global order. Instead, the validity of an execution is determined by a *memory consistency model*, which can be represented as a pair of constraints: emptiness and irreflexivity constraints over event orders [KLV23].

To keep the presentation manageable, we consider Multicopy-Atomic (MCA) models. It remains our future work to integrate the semantics of lock operations into non-MCA models. Under an MCA model, if a write event is visible to a thread that is not its issuing thread, then it is visible to all other threads as well. This property simplifies the axiomatic model because all cross-thread communications can be treated as global in MCA models. The rest of the section relies on MCA models that are defined in the following way.

An execution is consistent under an MCA model if

$$\text{irreflexive } (\text{ppo} \cup \texttt{com})^+$$
$$\text{irreflexive } (\texttt{po-loc} \cup \texttt{com})^+$$

where the ppo stands for *preserved program order*. Each MCA model provides its own definition of ppo. The second requirement corresponds to *SC-per-location*.

In addition, we omit treatment of Read-Modify-Write (RMW) events, and consider only fully fenced lock events. All of these could be accommodated without impacting the logic of our approach.

Under sequential consistency (SC), preserved program order is given by all $\texttt{po}$ orders.

$$\text{ppo} = \texttt{po}$$

Under x86-TSO [OSS09], preserved program order is given by

$$\text{ppo} = \text{po} \cap ((\text{Wrts} \times \text{Wrts}) \cup (\text{Rds} \times \text{Wrts}) \cup (\text{Rds} \times \text{Rds}))$$

Under ARMv8, preserved program order is given by the *locally-ordered-before* (lob) order from [ADG21].

We now augment these memory models with lock operations. We first define a new relation sync order. For each lock l, there is a linear order among the critical sections protected by l. For each two ordered critical sections of the same lock CS_1 and CS_2, where $CS_1 \rightarrow CS_2$, $\texttt{sync}_l$ is a relation from the release event of CS_1 to the acquire event of CS_2. sync is the union of $\texttt{sync}_l$ for all locks $l \in \mathsf{Locks}$.

We do not allow events to move into or out of a critical section. Therefore, each lock operation also has a fence-like effect. In practice, this is typically given by the use of fence instructions inside lock implementations. We augment preserved program order with the program orders where a lock operation occurs in-between. We use L to denote the union of lock acquires and releases.

$$\texttt{ppo} \cup (\texttt{po}; [\texttt{L}]) \cup ([\texttt{L}]; \texttt{po})$$

The most important property that locks should provide is mutual exclusion. Given two critical sections protected by the same lock, the events in one critical section should be all finished before the events in another critical section start. In other words, there should not be any interleaving among the events from two critical sections.

We can rule out this behavior by augmenting MCA models with:

$$\mathsf{irreflexive}\ (\texttt{ppo} \cup (\texttt{po}; [\texttt{L}]) \cup ([\texttt{L}]; \texttt{po}) \cup \texttt{com} \cup \texttt{sync})^{+}$$

It's easy to see that moving events into a critical section while preserving all other program orders does not introduce new behavior as it only monotonically adds more ppo order into the execution graph.

Finally, there may be open critical sections in an execution graph. By well-formedness of an execution graph, there is at most one open critical section per lock. The open critical sections should always be ordered as the last critical section in the linear order of that lock. Therefore, we add one more restriction to the memory model. For each lock l, if there is an open critical section, let $\mathsf{acq}(l)$ be the acquire event of that open critical section. The for every release event $\mathsf{rel}(l)$ of the same lock occur in the same execution, we have $\langle \mathsf{rel}(l), \mathsf{acq}(l) \rangle \in \texttt{sync}$.

Overall, we have the following definition for an MCA memory model augmented with lock operations.

▶ Definition 18 ($\mathcal{M}$-consistency)
Given a definition of preserved program order ppo from an memory model $\mathcal{M}$ for an MCA architecture, a complete execution graph G_ρ is $\mathcal{M}$-consistent if

- $(\texttt{po-loc} \cup \texttt{com})^+$ is irreflexive, and
- $(\texttt{ppo} \cup (\texttt{po}; [\mathsf{L}]) \cup ([\mathsf{L}]; \texttt{po}) \cup \texttt{com} \cup \texttt{sync})^+$ is irreflexive, and
- for each lock $l \in \mathsf{Locks}$, if $G_\rho.Open(l) = \mathsf{acq}(l)$, then for each release event $\mathsf{rel}(l) \in G_\rho.\mathsf{Rels}$, $\langle \mathsf{rel}(l), \mathsf{acq}(l) \rangle \in G_\rho.\texttt{sync}$.

As a sanity check, we prove a property that is a weaker variation of the DRF-SC theorem. The following proposition states that, if every pair of conflicting access is protected by some lock, which means there is no data race in the program, then the program is guaranteed to be sequentially consistent.

▶ Proposition 3
Let P be a program. For each sequentially consistent execution graph of P, if for each conflicting memory event $e_1 \bowtie e_2$, $e_1, e_2 \in CS_l$ for some $l \in \mathsf{Locks}$, then every sound execution of P is sequentially consistent.

Given the definition of soundness, in the rest of this paper, we explain how one can prove that a given predictive analysis is sound.

4.5 A Recipe to Prove Soundness

The soundness definition from §4.4 has an existential quantifier over witness execution graphs. Hence, a proof of soundness for a given predictive analysis should provide a scheme to construct a witness execution graph for each reported concurrent bug and show that the witness satisfies each of the four requirements of soundness.

Note that the soundness definition asks for a complete execution graph where the value of each event is concrete. In some cases where the `rf` order of the witness execution is altered from that of the input execution, the values of some events are not computable without knowing the program source code. On the other hand, there is a subset of events whose values are critical for the execution control flow, and therefore have to be preserved. For events that do not affect the control flow, their concrete values are unimportant to the soundness of the algorithm. Therefore, we use a *symbolic* execution graph as an intermediate form, which allows the values of a subset of events to be symbolic if they do not need to be preserved before memory orders are inserted.

To prove soundness for a predictive analysis, a witness execution can be constructed in the following steps:

§4.5.1 **Constructing a Symbolic Plain Execution.** Construct a symbolic plain execution graph $\hat{G}_\rho$ with an event map $\delta : \hat{G}_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$, such that $\hat{G}_\rho \rhd b$, and $\hat{G}_\rho$ is well-formed and executable.

§4.5.2 **Inserting Consistent Memory Orders.** Insert `rf`, `co`, and `sync` memory orders so that $\hat{G}_\rho$ is $\mathcal{M}$-consistent up to concrete events and well-formed.

§4.5.3 **Mapping to a Concrete Execution.** Map the symbolic execution $\hat{G}_\rho$ to a concrete execution graph G_ρ with a complete `rf`-map while preserving all the properties.

In the rest of this section, we discuss each step in details. The proofs of the propositions and lemmas can be found in Appendix O.

4.5.1 Constructing a Symbolic Plain Execution

Recall that a symbolic plain execution graph $\hat{G}_\rho$ is a tuple $(\mathsf{Evts}, \mathsf{po})$ where some of the events have symbolic values. To ensure executability, the program order po from G_σ has to be preserved in $\hat{G}_\rho$. Therefore, the task of constructing $\hat{G}_\rho$ is essentially finding a set of events to be included in $\hat{G}_\rho$.Evts and determine the concrete values of a subset of events in the set.

The set of events Evts to be included in the witness execution is determined by dependencies and lock semantics. We identify two types of dependencies, *control* and *data* dependencies. While modern architectures define other types of dependencies as well, control and data dependencies are two most fundamental dependencies that determines the soundness of a predictive analysis in the language of this paper.

- **Control dependencies**. Control dependencies determine the control flow of the program. Formally, it is a subset of po whose domain is a set of read events. For each $\langle r, e \rangle \in \texttt{ctrl}$ in an execution, the value of r determines whether the instruction that generates e is eventually executed at some point. In each execution state, the field $ctrl$ is a set of read events that is used to compute control dependency.

- **Data dependencies**. Data dependencies determine the data flow of the program. Formally, it is a subset of po whose domain is a set of read events and whose range is a set of write events. For each $\langle r, w \rangle \in \texttt{data}$ in an execution, the value of r determines the value of w.

The two dependencies encapsulate the sequential and control properties from any memory model considered in this paper. The two dependencies are sufficient to determine a set of events in the input execution that each bug sequence depends on. In other words, there is a set of events $S_\sigma \subseteq G_\sigma$.Evts with a subset $C_\sigma \subseteq S_\sigma$ such that the existence of events in S_σ and the values of the events in C_σ determine the control flow of the execution that leads to the bug sequence. These events can then be used to construct a plain execution graph, i.e. the witness execution. To ensure the witness execution satisfy the first three soundness requirements, S_σ and C_σ have to satisfy the following properties.

▶ Definition 19
Let $\langle S_\sigma, C_\sigma \rangle$ be two event sets such that $C_\sigma \subseteq S_\sigma \subseteq G_\sigma.\mathsf{Evts}$. We say $\langle S_\sigma, C_\sigma \rangle$ *enables* a bug sequence b if

I. For each event $e \in b$, if $\langle e', e \rangle \in G_\sigma.\mathsf{po}$, then $e' \in S_\sigma$

II. S_σ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$

III. S_σ is lock-feasible

IV. For each event $e \in b$, if $\langle r, e \rangle \in G_\sigma.\texttt{ctrl}$, then $e' \in C_\sigma$

V. For each $e \in S_\sigma$, if $\langle r, e \rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$

VI. For each $e \in C_\sigma$, if $\langle r, e \rangle \in G_\sigma.\texttt{data}$, then $r \in C_\sigma$

VII. For each $r \in C_\sigma$, there exists a write $w \in C_\sigma$ such that $r.val = w.val$ and $r.loc = w.loc$

VIII. $S_\sigma \cap b = \varnothing$

Given such a pair of event sets $\langle S_\sigma, C_\sigma \rangle$, one can construct a plain execution graph $\hat{G}_\rho = (\mathsf{Evts}, \mathsf{po})$ by the following steps. Let S_ρ be an event set such that $S_\rho \approx S_\sigma$ and $C_\sigma \subseteq S_\rho$. In other words, there is a bijective event map $\delta : S_\rho.\mathsf{Evts} \to S_\sigma.\mathsf{Evts}$ where $\delta(e) \approx e$ for each $e \in S_\rho$ and $\delta(e) = e$ for each $e \in C_\sigma$.

1. $\hat{G}_\rho.\mathsf{Evts} = S_\rho$.

2. $\hat{G}_\rho.\mathsf{po} = \delta^{-1}(G_\sigma.\mathsf{po} \cap (\delta(\hat{G}_\rho.\mathsf{Evts} \times \hat{G}_\rho.\mathsf{Evts})))$.

Given a pair of event set $\langle S_\sigma, C_\sigma \rangle$ that enables the bug sequence b, the constructed witness plain execution $\hat{G}_\rho$ can be shown to satisfy the first three soundness requirements.

▶ Proposition 4
If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, then $\hat{G}_\rho$ is well-formed up to concrete events and $\hat{G}_\rho \rhd b$.

▶ Proposition 5
If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, and $G_\sigma \in [\![P]\!]$, then $\hat{G}_\rho \in [\![P]\!]$.

Since the soundness of the witness execution depends on the pair $\langle S_\sigma, C_\sigma \rangle$, one essentially has to provide such a pair and show that it enables the reported bug sequence.

In practice, the precise information of control and data dependencies are rarely known to the predictive analysis. In addition, if the analysis does not record the concrete values of events in the input execution, then it'd be hard to determine which write event has the same value of a read event, as required by one of the conditions above. One way to overcome these challenges is tracing the $\mathsf{po} \cup \mathtt{rf}$ orders of the input execution G_σ and leverage the well-formed properties that they provide. The following lemma shows that any plain execution graphs of which the event set is downward-closed with respect to po and a subset of rf is read-feasible and executable.

▶ LEMMA 5
Let $\hat{G}_\rho = (\mathsf{Evts}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evts}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{rf}|_C)$ where C is the set of concrete read and write events of $\hat{G}_\rho$ such that $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ for a bug sequence b, and $\delta(e) = e$ for each $e \in C$. Then $\hat{G}_\rho$ is read-feasible up to C and executable.

LEMMA 5 does not guarantee lock feasibility. Lock feasibility may be ensured by tracing a partial order from G_σ, as stated in the following lemma.

▶ LEMMA 6
Let $\hat{G}_\rho = (\mathsf{Evts}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evts}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{sync})$, then $\hat{G}_\rho$ is lock-feasible.

As we will see in §4.6, analyses that over-approximate the control and data dependencies can apply LEMMA 5 and LEMMA 6 to show the well-formedness and the executability requirements.

4.5.2 Inserting Consistent Memory Orders

After a symbolic plain execution graph is determined, the next step is providing a memory order insertion scheme so that the complete execution graph is memory consistent. The goal in this step is to obtain an $\mathcal{M}$-consistent symbolic graph such that rf-map is defined and well-formed for all concrete read events, co is a total order among all write events to the same location, and sync is well-formed for each lock. While the insertion scheme is, in general, specific to the predictive algorithm, in some cases memory orders of G_σ can be reused since it is sequentially consistent, which automatically ensures $\mathcal{M}$-consistency.

We begin by inserting `rf` orders among concrete events in $\hat{G}_\rho$. For each concrete read in $\hat{G}_\rho$, by Lemma 4, we know that its value is inherited from the correspondent event in G_σ. From (VII) of Definition 19, there is a write event whose value is also preserved. Therefore, we can insert the same `rf` order between the preserved write and preserved reads.

Moreover, note that the only case where a cycle would potentially occur is when critical sections are reordered, due to the second requirement for $\mathcal{M}$-consistency: open critical sections should be ordered after all other critical sections of the same lock. Hence, if any of the events occur in the bug sequence is in an open critical section, all other critical sections of the same lock in $\hat{G}_\rho$ have to be ordered before the open critical section. For critical sections that originally occurred after some events from the bug sequence, it means they would need to be reordered with the bug events in $\hat{G}_\rho$. In order to show that $\hat{G}_\rho$ is $\mathcal{M}$-consistent, one has to show that such reordering can never cause a cycle that is forbidden by $\mathcal{M}$ to occur.

On the other hand, if the set of events in $\hat{G}_\rho$ is guaranteed *not* to include such critical sections, then the memory orders from G_σ can be reused and the resulting execution graph is still sequentially consistent. The following lemma demonstrates this idea.

▶ Lemma 7
Let G_σ be an input execution such that G_σ is sequentially consistent, equipped with a linear `trace` order. Let $\hat{G}_\rho$ be a symbolic plain execution that is well-formed, and $\hat{G}_\rho \rhd b$ where b is a reported bug. If for all acquire event $\mathsf{acq}(l) \in \hat{G}_\rho.\mathsf{Evts}$ such that $l \in \mathsf{LocksHeld}(e)$, $\langle \mathsf{acq}(l), e\rangle \in G_\sigma.\mathtt{trace}$ for each event $e \in b.\mathsf{Evts}$ that is in a critical section where the acquire event of the critical section $\mathsf{acq}(l) \in \hat{G}_\rho$, then there exists a memory order insertion scheme over $\hat{G}_\rho.\mathsf{Evts}$ such that $\hat{G}_\rho$ is sequentially consistent.

As we will see in §4.6, predictive analyses that preserve synchronization orders [MPV21, MKV18] use this lemma to insert memory orders when constructing the witness execution.

4.5.3 Mapping to a Concrete Execution

Finally, once the memory orders are inserted while maintaining $\mathcal{M}$-consistency, the following lemma shows that there exists a well-formed concrete graph, i.e., G_ρ, that can be obtained by concretizing the symbolic events in $\hat{G}_\rho$ and all properties from the previous steps are preserved.

▶ Lemma 8
Let $\hat{G}_\rho$ be a symbolic execution with a well-formed `rf`-map over concrete events, a total `co` order over write events to the same location, and a well-formed `sync` over lock events. If $\hat{G}_\rho$ is $\mathcal{M}$-consistent and $\hat{G}_\rho \in [\![P]\!]$ with $e.val \in \mathsf{Val}$ for each $e \in \mathsf{preserve}(b)$, then there exists a map $\Theta : \mathsf{Sym} \to \mathsf{Val}$ such that the concrete execution $G_\rho = \Theta(\hat{G}_\rho)$ with a complete `rf`-map is $\mathcal{M}$-consistent and $G_\rho \in [\![P]\!]$.

The result of applying this lemma is a complete and concrete execution graph that satisfies the four requirements of soundness, which finishes the soundness proof.

4.6 Proving Race Prediction Algorithms Sound

In §4.2, we reviewed a set of existing predictive analyses that used various soundness definitions as their correctness criteria. Fig. 4.7 shows six of the nine analyses from §4.2 that focus on predicting data races with their race reporting criteria. Because the soundness definitions used in their papers were different, their soundness proofs are hard to compare with each other. In this section, we take a closer look at these algorithms and use our recipe on each of them to show their soundness.

In the rest of the section, we assume $e_1, e_2 \in G_\sigma$ are two events from the input execution and $e_1 \bowtie e_2$. In addition, to keep the discussions concise, $\langle e_1, e_2\rangle \in G_\sigma$.`trace`.

4.6.1 M2

M2 [Pav20] determines data races by computing linearizable closures. In particular, a set of events, $\mathsf{RCone}_\sigma(e_1, e_2)$, is first computed. Then a partial order is inserted among the events in

	Analyses	Race Reporting Criteria
[Pav20]	M2	$\mathtt{P}$ is a strict partial order over $\mathsf{RCone}_\sigma(e_1, e_2)$
[HMR14]	RVPREDICT	$\exists \rho \models \Phi^\sigma_{\mathsf{mhb}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$
[HH16]	MCR-TSO	$\exists \rho \models \Phi^\sigma_{\mathsf{ppo}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$
[FF09]	HB	the first $e_1 \parallel_{\mathtt{hb}} e_2$
[MKV18]	SHB	$e_1 \parallel_{\mathtt{shb}} e_2$
[MPV21]	SYNCP	$\{e_1, e_2\} \cap \mathsf{SPIdeal}_\sigma(e_1, e_2) = \varnothing$
§P	ENHANCED-MCR-TSO	$\exists \rho \models \Phi^\sigma_{\mathsf{ppo}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$ after Read Elimination

Figure 4.7: Race Reporting Criterion of Various Race Prediction Algorithms

this set until a linearizable state is reached. If e_1 or e_2 is included in $\mathsf{RCone}_\sigma(e_1, e_2)$, or a cycle occurs during the process of inserting the partial order, then $\langle e_1, e_2 \rangle$ is not a data race. Otherwise, $\langle e_1, e_2 \rangle$ is reported as a data race. Formally, $\mathsf{RCone}_\sigma(e_1, e_2)$ is defined inductively as the following:

- $\{\mathsf{prev}_\sigma(e_1), \mathsf{prev}_\sigma(e_2)\} \subseteq \mathsf{RCone}_\sigma(e_1, e_2)$, where $\langle \mathsf{prev}_\sigma(e), e \rangle \in G_\sigma.\mathsf{po}|_{imm}$ for all $e \in G_\sigma.\mathsf{Evts}$,
- for each event $e \in G_\sigma.\mathsf{Evts}$, if $\langle e, e' \rangle \in G_\sigma.(\mathsf{po} \cup \mathtt{rf})$ for some event $e' \in \mathsf{RCone}_\sigma(e_1, e_2)$, then $e \in \mathsf{RCone}_\sigma(e_1, e_2)$,
- for each acquire event $\mathsf{acq}(l) \in \mathsf{RCone}_\sigma(e_1, e_2)$, if $\mathsf{acq}(l).tid \neq e_1.tid$ and $\mathsf{acq}(l).tid \neq e_2.tid$, then there is a release event $\mathsf{rel}(l) \in \mathsf{RCone}_\sigma(e_1, e_2)$ such that $\mathsf{match}(\mathsf{rel}(l)) = \mathsf{acq}(l)$.

Then a closure algorithm is applied over events in $\mathsf{RCone}_\sigma(e_1, e_2)$. The algorithm inserts a strict partial order $\mathtt{P}$ based on the following closure rules.

1. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})|_{\mathsf{RCone}} \subseteq \mathtt{P}$,
2. For every $\mathsf{acq}(l) = Open_l(\mathsf{RCone}_\sigma(e_1, e_2))$ and every $\mathsf{rel}(l) \in \mathsf{RCone}_\sigma(e_1, e_2).\mathsf{Rels}$, $\mathsf{rel}(l) \xrightarrow{\mathtt{P}} \mathsf{acq}(l)$,

3. If $w' \xrightarrow{\texttt{P}} r$ and $w \xrightarrow{\texttt{rf}} r$ then $w' \xrightarrow{\texttt{P}} w$ for each $w, w' \in \mathsf{RCone}_\sigma(e_1, e_2).\mathsf{Wrts}$ and $r \in \mathsf{RCone}_\sigma(e_1, e_2).\mathsf{Rds}$ where $w'.loc = w.loc = r.loc$

4. If $w \xrightarrow{\texttt{P}} w'$ and $w \xrightarrow{\texttt{rf}} r$ then $r \xrightarrow{\texttt{P}} w'$ for each $w, w' \in \mathsf{RCone}_\sigma(e_1, e_2).\mathsf{Wrts}$ and $r \in \mathsf{RCone}_\sigma(e_1, e_2).\mathsf{Rds}$ where $w'.loc = w.loc = r.loc$

5. For $\mathsf{acq}_1(l) = \mathsf{match}(\mathsf{rel}_1(l)), \mathsf{acq}_2(l) = \mathsf{match}(\mathsf{rel}_2(l))$, if $\mathsf{acq}_1(l) \xrightarrow{\texttt{P}} \mathsf{rel}_2(l)$, then $\mathsf{rel}_1(l) \xrightarrow{\texttt{P}} \mathsf{acq}_2(l)$

Moreover, if there exists any pair of events $e \bowtie e' \in \mathsf{RCone}_\sigma(e_1, e_2)$ such that $e_i.tid \notin \{e.tid, e'.tid\}$ for a non-deterministically chosen $i \in \{1, 2\}$, $e \xrightarrow{\texttt{trace}} e'$, and $e \parallel_{\texttt{P}} e'$, then $\langle e, e' \rangle$ is added into $\texttt{P}$ and the closure rules above are applied to reach a fixed point.

The soundness of M2 is stated as the following.

▶ Theorem 4
If $\mathsf{LocksHeld}(e_1) \cap \mathsf{LocksHeld}(e_2) = \varnothing$, $\{e_1, e_2\} \cap \mathsf{RCone}_\sigma(e_1, e_2) = \varnothing$, and $\texttt{P}$ computed as described by closure rule 1-5 is a strict partial order such that $\forall \bar{e_1}, \bar{e_2} \in \mathsf{RCone}_\sigma(e_1, e_2) \backslash G_\sigma.\mathsf{Evts}|_{e_i.tid}$, $\bar{e_1} \bowtie \bar{e_2} \Rightarrow \bar{e_1} \nparallel \bar{e_2}$ where $i \in \{1, 2\}$, then $\langle e_1, e_2 \rangle$ is a sound data race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \mathsf{RCone}_\sigma(e_1, e_2) \quad C_\sigma = G_\sigma.(\mathsf{Rds} \cup \mathsf{Wrts}) \cap S_\sigma$$

From the definition, we know that S_σ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \texttt{rf})$. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evts} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ is the identity function.

We first show that $G_\rho \rhd b$ where $b = e_1 e_2$. To start with, $\{e_1, e_2\} \cap \mathsf{RCone}_\sigma(e_1, e_2) = \varnothing$ comes from the assumption. In addition, No Skipping is satisfied by the base condition of the definition, i.e., $\{\mathsf{prev}(e_1), \mathsf{prev}(e_2)\} \subseteq \mathsf{RCone}_\sigma(e_1, e_2)$. Since δ is the identity function, for each $r \in G_\sigma.\mathsf{Rds}$

such that $\langle r, e_1 \rangle \in G_\sigma.\texttt{ctrl}$ or $\langle r, e_2 \rangle \in G_\sigma.\texttt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathsf{Evts}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. Since $G_\rho.\mathsf{Evts}$ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \texttt{rf})$, by Lemma 5, G_ρ is read feasible and executable. From that last condition of RCone's definition, we know that every critical section on thread t, where $e_1.tid \neq t \neq e_2.tid$, is closed. Hence, open critical sections can only occur on thread t_1 and t_2 where $t_1 = e_1.tid$ and $t_2 = e_2.tid$. By well-formedness of G_σ, we know that at most one open critical section of each lock can occur on each thread. Given that $\mathsf{LocksHeld}(e_1) \cap \mathsf{LocksHeld}(e_2) = \varnothing$, we can conclude that at most one open critical section is included in $\mathsf{RCone}_\sigma(e_1, e_2)$ for each lock. Thus, $\mathsf{RCone}_\sigma(e_1, e_2)$ is lock feasible.

▶ *Inserting Memory Orders.* Lastly, we show that there exists a memory order insertion scheme such that G_ρ is sequentially consistent. First, we set $G_\rho.(\mathsf{po} \cup \texttt{rf}) = G_\sigma.(\mathsf{po} \cup \texttt{rf})|_{\mathsf{RCone}}$. From 1., we know that $G_\rho.(\mathsf{po} \cup \texttt{rf}) \subseteq \texttt{P}$. Next, for every $\mathsf{acq}(l) = Open_l(\mathsf{RCone}_\sigma(e_1, e_2))$ and every $\mathsf{rel}(l) \in \mathsf{RCone}_\sigma(e_1, e_2)$, we set $\langle \mathsf{rel}(l), \mathsf{acq}(l) \rangle \in G_\rho.\texttt{sync}$. We call this subset of $\texttt{sync}$ order as $\texttt{sync}|_{open}$. Then from 2., it's easy to see that $\texttt{sync}|_{open} \subseteq \texttt{P}$. In addition, the third requirement for $\mathcal{M}$-consistency in Definition 18, where $\mathcal{M}$ is sequential consistency, is satisfied. We now insert $\texttt{co}$ orders.

For each pair of write events accessing the same location, $\langle w_1, w_2 \rangle$, $\texttt{co}$ is inserted in the following way,

- if $w_1 \xrightarrow{\texttt{P}} w_2$, then $w_1 \xrightarrow{\texttt{co}} w_2$; if $w_2 \xrightarrow{\texttt{P}} w_1$, then $w_2 \xrightarrow{\texttt{co}} w_1$
- otherwise, if $w_1.tid = e_i.tid$, then $w_1 \xrightarrow{\texttt{co}} w_2$; if $w_2.tid = e_i.tid$, then $w_2 \xrightarrow{\texttt{co}} w_1$

For each $\mathsf{rel}_1(l)$, $\mathsf{rel}_2(l)$ and $\mathsf{acq}_1(l)$, $\mathsf{acq}_2(l)$ such that $\mathsf{match}(\mathsf{rel}_1(l)) = \mathsf{acq}_1(l)$, and $\mathsf{match}(\mathsf{rel}_2(l)) = \mathsf{acq}_2(l)$, $\texttt{sync}$ is inserted in the following way,

- if $\mathsf{rel}_1(l) \xrightarrow{\texttt{P}} \mathsf{acq}_2(l)$, then $\mathsf{rel}_1(l) \xrightarrow{\texttt{sync}} \mathsf{acq}_2(l)$; if $\mathsf{rel}_2(l) \xrightarrow{\texttt{P}} \mathsf{acq}_1(l)$, then $\mathsf{rel}_2(l) \xrightarrow{\texttt{sync}} \mathsf{acq}_1(l)$,

- otherwise, if $\mathsf{rel}_1(l).tid = e_i.tid$ then $\mathsf{rel}_1(l) \xrightarrow{\mathtt{sync}} \mathsf{acq}_2(l)$; if $\mathsf{rel}_2.tid = e_i.tid$, then $\mathsf{rel}_2(l) \xrightarrow{\mathtt{sync}} \mathsf{acq}_1(l)$

Note that after inserting the ordered as described above, $\mathtt{co}$ and $\mathtt{sync}$ are well-formed in G_ρ.

We now show that this insertion scheme guarantees sequential consistency. Suppose, towards contradiction, that there is a $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ cycle in G_ρ after we finish inserting the orders. First, since $(\mathsf{po} \cup \mathtt{rf}) \subseteq \mathtt{P}$ and $\mathtt{P}$ is a strict partial order, one of the edges forming this cycle must be a $(\mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})$ edge from some event e_a to e_b such that $e_a.tid = e_i.tid \neq e_b.tid$ and $e_a \parallel_{\mathtt{P}} e_b$. In addition, there is also a $\mathtt{P}$ edge in this cycle from an event e_c to some event e_d such that $e_d.tid = e_i.tid \neq e_c.tid$. Since they form a cycle, we can infer that $e_d \xrightarrow{\mathsf{po}}{}^* e_a$, which also means $e_d \xrightarrow{\mathtt{P}}{}^* e_a$. In other words, the cycle is structure as

$$e_d \xrightarrow{\mathtt{P}}{}^* e_a \xrightarrow{\mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync}} e_b \xrightarrow{\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync}}{}^* e_c \xrightarrow{\mathtt{P}} e_d$$

Since $e_b.tid \neq e_i.tid \neq e_c.tid$, for any conflicting events $e \bowtie e'$, we know that $e \nparallel_{\mathtt{P}} e'$. Given this and that $(\mathsf{po} \cup \mathtt{rf}) \subseteq \mathtt{P}$, the $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ path between e_b and e_c must also be a $\mathtt{P}$ path (note that all communication edges relates conflicting events), i.e., $e_b \xrightarrow{\mathtt{P}}{}^* e_c$. Now we analyze each possible case.

- $e_d \xrightarrow{\mathtt{P}}{}^* e_a \xrightarrow{\mathtt{co}} e_b \xrightarrow{\mathtt{P}}{}^* e_c \xrightarrow{\mathtt{P}} e_d$. But $e_b \xrightarrow{\mathtt{P}} e_a$ implies that the $e_b \xrightarrow{\mathtt{co}} e_a$, which contradicts with $e_a \xrightarrow{\mathtt{co}} e_b$.

- $e_d \xrightarrow{\mathtt{P}}{}^* e_a \xrightarrow{\mathtt{fr}} e_b \xrightarrow{\mathtt{P}}{}^* e_c \xrightarrow{\mathtt{P}} e_d$. Then there exists a write event $w \xrightarrow{\mathtt{rf}} e_a$ and $w \xrightarrow{\mathtt{co}} e_b$. From $e_b \xrightarrow{\mathtt{P}} e_a$ and $w \xrightarrow{\mathtt{rf}} e_a$, we know $e_b \xrightarrow{\mathtt{P}} w$ because of the closure rule 3., which contradicts with $w \xrightarrow{\mathtt{co}} e_b$.

- $e_d \xrightarrow{\mathtt{P}}{}^* e_a \xrightarrow{\mathtt{sync}} e_b \xrightarrow{\mathtt{P}}{}^* e_c \xrightarrow{\mathtt{P}} e_d$. Then $e_a \in G_\rho.\mathsf{Rels}$, $e_b \in G_\rho.\mathsf{Acqs}$, and there exists $\mathsf{acq}_a = \mathsf{match}(e_a)$ and $\mathsf{match}(\mathsf{rel}_b) = e_b$. By the closure rule 5., we have $\mathsf{rel}_b \xrightarrow{\mathtt{sync}} \mathsf{acq}_a$, which contradicts with $e_a \xrightarrow{\mathtt{sync}} e_b$.

Since each case gives us a contradiction, we can conclude that $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ is irreflexive. That is, G_ρ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

4.6.2 RVPredict

RVPredict [HMR14] is an SMT-based approach to predicting data races in a program. Given an input execution G_σ with a `trace` order, the algorithm maps each event from the input execution to an integer variable that represents its order in a potential witness along with a formula generated from the input execution. A pair of conflicting events is reported as a data race if the set of constraints is satisfiable, i.e., there exists a map from the variables to integers that solves the contraints. Formally, for each event $e \in G_\sigma.\mathsf{Evts}$, $O_e \in \mathcal{O}$ is an order variable for e. A formula Φ is generated from G_σ:

$$\Phi = \Phi_{\mathsf{mhb}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}$$

where each sub-formula is defined as the following.

$$\Phi_{\mathsf{mhb}} = \bigwedge_{\langle e,e'\rangle \in G_\sigma.\mathsf{po}} O_e < O_{e'}$$

$$\Phi_{\mathsf{lock}} = \bigwedge_{\mathsf{rel}_1(l),\mathsf{rel}_2(l) \in G_\sigma.\mathsf{Rels}} \left(O_{\mathsf{rel}_1(l)} < O_{\mathsf{acq}_2(l)} \vee O_{\mathsf{rel}_2(l)} < O_{\mathsf{acq}_1(l)}\right)$$

$$\text{where } \mathsf{acq}_i(l) = \mathsf{match}(\mathsf{rel}_i(l))$$

$$\Phi_{\mathsf{race}} = (O_{e_2} - O_{e_1} = 1) \wedge \Phi^{\approx}_{\mathsf{cf}}(e_1) \wedge \Phi^{\approx}_{\mathsf{cf}}(e_2)$$

$$\Phi^{\approx}_{\mathsf{cf}}(e) = \Phi_{\mathsf{cf}}(br) \text{ where } br \in G_\sigma.\mathsf{Brs} \text{ is the last branch event such that } \langle br, e\rangle \in G_\sigma.\mathsf{po}$$

$$\Phi_{\mathsf{cf}}(e) = \bigwedge_{r \in G_\sigma.\mathsf{Rds}} \Phi_{\mathsf{cf}}(r) \text{ where } \langle r, e\rangle \in G_\sigma.\mathsf{po} \text{ and } e \in G_\sigma.(\mathsf{Brs} \cup \mathsf{Wrts})$$

$$\Phi_{\mathsf{cf}}(e) = \Phi_{\mathsf{cf}}(w) \land O_w < O_e \bigwedge_{w' \in G_\sigma.\mathsf{Wrts}} (O_{w'} < O_w \lor O_e < O_{w'})$$

where $w.loc = e.loc = w'.loc$, $\langle w, e\rangle \in G_\sigma.\texttt{rf}$, $w \neq w'$, and $e \in G_\sigma.\mathsf{Rds}$

The soundness theorem for RVPredict is the following.

▶ Theorem 5
If there exists a map $\rho : \mathcal{O} \to \mathsf{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\mathsf{mhb}} \land \Phi_{\mathsf{lock}} \land \Phi_{\mathsf{race}}(e_1, e_2)$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a subset of events defined as the following.

$$S_\sigma = \{e \in G_\sigma \mid \rho(O_e) < \rho(O_{e_1})\} \quad C_\sigma = R \cup W \text{ where}$$

$$R = \{r \in G_\sigma.\mathsf{Rds} \mid (\langle r, br\rangle \in G_\sigma.\mathsf{po} \text{ for some branch event } br \in S_\sigma.\mathsf{Brs})$$

$$\lor (\exists w \in C_\sigma, \langle r, w\rangle \in G_\sigma.\mathsf{po})\}$$

$$W = \{w \in G_\sigma.\mathsf{Wrts} \mid \exists r \in C_\sigma, \langle w, r\rangle \in G_\sigma.\texttt{rf}\}$$

Since $\rho(O_{e_2}) - \rho(O_{e_1}) = 1$, we know that $S_\sigma \cap \{e_1, e_2\} = \varnothing$.

By Φ_{mhb}, we have that S_σ is po-closed.

Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\mathsf{Evts} \to S_\sigma.\mathsf{Evts}$ such that $\delta(e) \approx e$. If there is a last branch event $br_i \in G_\sigma.\mathsf{Brs}$ that po-ordered before e_i, we have $br_i \in S_\sigma$ for $i \in \{1, 2\}$. Then for all read event such that $\langle r, br_i\rangle \in G_\sigma.\mathsf{po}$ (note that $r \in S_\sigma$), set $\delta^{-1}(r) = r$. For each write event $w \in S_\sigma.\mathsf{Wrts}$, if for all $\langle r, w\rangle \in G_\sigma.\mathsf{po}$ (note that $r \in S_\sigma$), $r = \delta^{-1}(r)$, then set $w = \delta^{-1}(w)$. For each read event $r \in S_\sigma.\mathsf{Rds}$, if there is $\langle w, r\rangle \in G_\sigma.\texttt{rf}$, then by $\Phi_{\mathsf{cf}}(r)$ we have $w \in S_\sigma$. If $\delta^{-1}(w) = w$, then set $\delta^{-1}(r) = r$. Otherwise, we assign a distinct symbolic value $\delta^{-1}(e).val = \hat{s}$ for read or write event $e \in S_\sigma$. Observe that the set of concrete events $C \subseteq S_\rho$ is $(\mathsf{po} \cup \texttt{rf})$-closed by construction.

Let $\hat{G}_\rho$ be a symbolic plain execution graph such that $\hat{G}_\rho.\mathsf{Evts} = S_\rho$ and $\hat{G}_\rho.\mathsf{po} = \delta(G_\sigma.\mathsf{po}|_{S_\sigma})$.

We first show that $\hat{G}_\rho \rhd b$ where $b = e_1e_2$. For $i \in \{1,2\}$ and each event $e \in G_\sigma$ such that $\langle e, e_i\rangle \in G_\sigma.\mathsf{po}$, from Φ_{mhb}, we know that $\rho(O_e) < \rho(O_{e_i})$. Hence $e \in S_\sigma$ and there is an event $e' \in \hat{G}_\rho$ such that $\delta(e') = e$. Therefore, No Skipping is satisfied. In addition, if $\langle r, e_i\rangle \in G_\sigma$, then there is a branch event $\langle br_i, e_i\rangle, \langle r, br_i\rangle \in G_\sigma.\mathsf{po}$. From the definition of δ above, $\delta^{-1}(r) = r$. Hence there is $r' \in \hat{G}_\rho.\mathsf{Rds}$ such that $\delta(r') = r = r'$ and $r' \in C$. Therefore, Same Control Flow is satisfied.

We now show that $\hat{G}_\rho$ is well-formed and executable. Observe that $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ in $\hat{G}_\rho$ since $\mathtt{ctrl} \subseteq \mathsf{po}; [\mathsf{Brs}]; \mathsf{po}$ and $\mathtt{data} \subseteq \mathsf{po}$. By Lemma 5, $\hat{G}_\rho$ is read feasible and executable. For each lock l, if two acquire events $\mathsf{acq}_1(l), \mathsf{acq}_2(l) \in S_\sigma$, then by Φ_{lock}, either $\mathsf{rel}_1(l) \in S_\sigma$ or $\mathsf{rel}_2 \in S_\sigma$. Since $\hat{G}_\rho.(\mathsf{Acqs} \cup \mathsf{Rels}) = S_\sigma.(\mathsf{Acqs} \cup \mathsf{Rels})$, we have if two acquire events $\mathsf{acq}_1(l), \mathsf{acq}_2(l) \in \hat{G}_\rho.\mathsf{Acqs}$, then either $\mathsf{rel}_1(l) \in \hat{G}_\rho.\mathsf{Rels}$ or $\mathsf{rel}_2 \in \hat{G}_\rho.\mathsf{Rels}$. Therefore, $\hat{G}_\rho$ is lock feasible.

▶ *Inserting Memory Orders.* We now show that there is a memory order insertion scheme such that $\hat{G}_\rho$ is sequentially consistent up to C. For each read event $r \in C \subseteq \hat{G}_\rho.\mathsf{Rds}$, from the definition of δ, we know that there is a write event $w \in C$ such that $\langle \delta(w), \delta(r)\rangle \in G_\sigma.\mathtt{rf}$. Insert $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{rf}$. For each write events $w, w' \in \hat{G}_\rho.\mathsf{Wrts}$, if $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$, then $\langle w, w'\rangle \in \hat{G}_\rho.\mathtt{co}$. For each lock l, if $\rho(O_{\mathsf{rel}(l)}) < \rho(O_{\mathsf{acq}(l)})$ for some release event $\mathsf{rel}(l)$ and acquire event $\mathsf{acq}(l)$, then $\langle \mathsf{rel}(l), \mathsf{acq}(l)\rangle \in \hat{G}_\rho.\mathtt{sync}$. Since ρ maps order variables to integers, which are linearly ordered, $\hat{G}_\rho.\mathtt{co}$ is a linear among writes to the same location and $\hat{G}_\rho.\mathtt{sync}$ is well-formed. We argue $\hat{G}_\rho$ is sequentially consistent after inserting the orders. First, from Φ_{lock}, we can infer that open critical sections are ordered last. For each $\langle r, w'\rangle \in \hat{G}_\rho.\mathtt{fr}$, there is a write w such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{rf}$ and $\langle w, w'\rangle \in \hat{G}_\rho.\mathtt{co}$. By the order insertion scheme above, it means $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$. From $\Phi_{\mathsf{cf}}(r)$, we can infer that $\rho(O_{\delta(r)}) < \rho(O_{\delta(w')})$. From Φ_{mhb}, for each $\langle e, e'\rangle \in \hat{G}_\rho.\mathsf{po}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. From $\Phi_{\mathsf{cf}}(r)$, for each $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{rf}$, we have $\rho(O_{\delta(w)}) < \rho(O_{\delta(r)})$. Therefore, for any $\langle e, e' \rangle \in \hat{G}_\rho.(\text{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co} \cup \texttt{sync})^+$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. Thus, $\hat{G}_\rho$ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Lastly, by Lemma 8, there exists a concrete execution G_ρ inherits all the properties of $\hat{G}_\rho$, i.e., $G_\rho \rhd b$, and G_ρ is well-formed, executable, and sequentially consistent. □

4.6.3 MCR-tso

Here we provide a formal proof for a race predictor built based on the constraint encoding from [HH16]. While the paper of MCR [HH16] did not provide any example of new data races discovered under TSO, we observe that the non-SC-race example from Fig. 8 of [Pav20] is such a data race. We provide a detailed explanation of the example in Appendix Q.

Given an input execution G_σ with a $\texttt{trace}$ order, the algorithm maps each event from the input execution to an integer variable that represents its order in a potential witness along with a formula generated from the input execution. A pair of conflicting events is reported as a data race if the set of constraints is satisfiable, i.e., there exists a map from the variables to integers that solves the constraints. Formally, for each event $e \in G_\sigma.\text{Evts}$, $O_e \in \mathcal{O}$ is an order variable for e. A formula Φ is generated from G_σ:

$$\Phi = \Phi_{\text{ppo}} \wedge \Phi_{\text{lock}} \wedge \Phi_{\text{race}}$$

where each sub-formula is defined as the following.

$$\begin{aligned}
\Phi_{\text{ppo}} = & \bigwedge_{\langle e,e' \rangle \in G_\sigma.\text{ppo}} O_e < O_{e'} \quad \bigwedge_{\langle e,e' \rangle \in G_\sigma.\texttt{po-loc}} O_e < O_{e'} \\
& \bigwedge_{\langle e,\text{rel}(l) \rangle, \langle \text{acq}(l),e' \rangle \in G_\sigma.\text{po}} O_e < O_{\text{rel}(l)} \wedge O_{\text{acq}(l)} < O_{e'} \\
\Phi_{\text{lock}} = & \bigwedge_{\text{rel}_1(l),\text{rel}_2(l) \in G_\sigma.\text{Rels}} \left(O_{\text{rel}_1(l)} < O_{\text{acq}_2(l)} \vee O_{\text{rel}_2(l)} < O_{\text{acq}_1(l)} \right)
\end{aligned}$$

where $\mathsf{acq}_i(l) = \mathsf{match}(\mathsf{rel}_i(l))$

$$\Phi_{\mathsf{race}} = (O_{e_2} - O_{e_1} = 1) \wedge \Phi_{\mathsf{obs}}$$

$$\Phi_{\mathsf{obs}} = \bigwedge_{\langle w,r\rangle \in G_\sigma.\mathtt{rf}} (O_w < O_r) \bigwedge_{w' \in G_\sigma.\mathsf{Wrts}} (O_{w'} < O_w \vee O_r < O_{w'})$$

where $w.loc = r.loc = w'.loc$, $w \neq w'$, and $r \in G_\sigma.\mathsf{Rds}$

The soundness theorem is the following.

▶ THEOREM 6
If there exists a map $\rho : \mathcal{O} \to \mathsf{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\mathtt{ppo}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}(e_1, e_2)$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a lock-feasible event set that is downward-closed w.r.t. $G_\sigma.\mathsf{po} \cup \mathtt{rf}$ from $b = e_1e_2$. Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\mathsf{Evts} \to S_\sigma.\mathsf{Evts}$ such that $\delta(e) \approx e$. Let $C_\sigma = G_\sigma.(\mathsf{Wrts} \cup \mathsf{Rds}) \cap S_\sigma$. If $\delta(e) \in C_\sigma$, then we set $\delta(e) = e$. Let $\hat{G}_\rho$ be a symbolic plain execution graph such that $\hat{G}_\rho.\mathsf{Evts} = S_\rho$ and $\hat{G}_\rho.\mathsf{po} = \delta(G_\sigma.\mathsf{po}|_{S_\sigma})$.

By LEMMA 5 and the fact that S_σ is lock-feasible, we get $\hat{G}_\rho$ is well-formed and executable. In addition, since S_σ is downward-closed from b, $\hat{G}_\rho \rhd b$.

▶ *Inserting Memory Orders.* We now provide a memory insertion scheme such that $\hat{G}_\rho$ is TSO-consistent. First, for each read event $r \in C_\sigma$, from the definition of δ and C_σ, we know that there exists a write event $w \in C_\sigma$ such that $\langle \delta(w), \delta(r)\rangle \in G_\sigma.\mathtt{rf}$. Insert $\langle w, r\rangle \in \hat{G}_\rho$. From Φ_{obs}, we know that $\rho(O_w) < \rho(O_r)$. For each write events $w, w' \in \hat{G}_\rho.\mathsf{Wrts}$, if $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$, then $\langle w, w'\rangle \in \hat{G}_\rho.\mathtt{co}$. For each lock l, if $\rho(O_{\mathsf{rel}(l)}) < \rho(O_{\mathsf{acq}(l)})$ for some release event $\mathsf{rel}(l)$ and acquire event $\mathsf{acq}(l)$, then $\langle \mathsf{rel}(l), \mathsf{acq}(l)\rangle \in \hat{G}_\rho.\mathtt{sync}$. Since ρ maps order variables to integers, which are linearly ordered, $\hat{G}_\rho.\mathtt{co}$ is a linear among writes to the same location and $\hat{G}_\rho.\mathtt{sync}$ is well-formed. We argue $\hat{G}_\rho$ is TSO-consistent after inserting the orders. First, from

Φ_{lock}, we can infer that open critical sections are ordered last. For each $\langle r, w' \rangle \in \hat{G}_\rho.\texttt{fr}$, there is a write w such that $\langle w, r \rangle \in \hat{G}_\rho.\texttt{rf}$ and $\langle w, w' \rangle \in \hat{G}_\rho.\texttt{co}$. By the order insertion scheme above, it means $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$. From Φ_{obs}, we have $\rho(O_{\delta(r)}) < \rho(O_{\delta(w')})$. From $\Phi_{\texttt{ppo}}$, for each $\langle e, e' \rangle \in \hat{G}_\rho.\texttt{ppo}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$, and for each $\langle e, e' \rangle \in \hat{G}_\rho.\texttt{po-loc}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. Hence, we can infer that $(\texttt{ppo} \cup (\texttt{po}; [\texttt{L}]) \cup ([\texttt{L}]; \texttt{po}) \cup \texttt{com} \cup \texttt{sync})^+$ and $(\texttt{po-loc} \cup \texttt{com} \cup \texttt{sync})^+$ are both irreflexive. That is, $\hat{G}_\rho$ is TSO-consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

4.6.4 Happens-Before (HB)

Happens-before (HB) is a partial order that is commonly used for partial-order-based dynamic race analyses. A partial-order-based race prediction algorithm predicts data races based on whether two conflicting events are ordered by a partial order built by the algorithm from the input execution. Formally, let D be a partial order built by the algorithm given an input trace σ. For an event pair $\langle e_1, e_2 \rangle \in G_\sigma.\text{Evts}$ such that $e_1 \bowtie e_2$, if $e_1 \parallel_D e_2$, then $\langle e_1, e_2 \rangle$ is reported as a predictable race.

The happens-before ($\texttt{hb}$) order is defined as the following.

$$\texttt{hb} = (\texttt{po} \cup \texttt{sync})^+$$

In addition, Happens-before only guarantees soundness of the first pair of conflicting events that is not ordered by $\texttt{hb}$. In other words, an extra assumption, i.e., every conflicting pair of events before e_1 and e_2 are ordered by $\texttt{hb}$, is added to its soundness theorem.

▶ THEOREM 7
If $\langle e_1, e_2 \rangle$ is the first reported race such that $e_1 \parallel_{\texttt{hb}} e_2$, i.e., for all events $e \xrightarrow{\texttt{trace}}_\sigma e' \xrightarrow{\texttt{trace}}_\sigma e_2$ such that $e \bowtie e'$, $\langle e, e' \rangle \in G_\sigma.\texttt{hb}$, then $\langle e_1, e_2 \rangle$ is a sound data race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \{e \in G_\sigma.\mathsf{Evts} \mid \langle e, e_1\rangle \in G_\sigma.\mathtt{hb} \ \vee \ \langle e, e_2\rangle \in G_\sigma.\mathtt{hb}\}$$

$$C_\sigma = G_\sigma.(\mathsf{Rds} \cup \mathsf{Wrts}) \cap S_\sigma$$

Because hb is transitive, S_σ is downward-closed w.r.t. hb. In addition, note that for each $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$ and $w \xrightarrow{\mathtt{trace}} r \xrightarrow{\mathtt{trace}} e_1$, since $w \bowtie r$ by definition, then $\langle w, r\rangle \in G_\sigma.\mathtt{hb}$. Similarly, for each $r \in G_\sigma.\mathsf{Rds}$ such that $\langle e_1, r\rangle \in G_\sigma.\mathtt{rf}$ and $e_1 \xrightarrow{\mathtt{trace}} r \xrightarrow{\mathtt{trace}} e_2$, $\langle e_1, r\rangle \in G_\sigma.\mathtt{hb}$. Hence, for every reads $r \in S_\sigma.\mathsf{Rds}$, if there is $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$, then $w \in S_\sigma$, i.e., S_σ is downward-closed w.r.t. $G_\sigma.\mathsf{po} \cup \mathtt{rf} \cup \mathtt{sync}$. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evts} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ is the identity function. Therefore, we omit the application of δ in the rest of the proof for better readability.

We first show that $G_\rho \rhd b$. Because $e_1 \parallel_{\mathtt{hb}} e_2$, we know that $\{e_1, e_2\} \cap S_\sigma = \varnothing$. Then No Skipping is satisfied because $G_\sigma.\mathsf{po} \subseteq G_\sigma.\mathtt{hb}$. Since δ is the identity function, for each $r \in G_\sigma.\mathsf{Rds}$ such that $\langle r, e_1\rangle \in G_\sigma.\mathtt{ctrl}$ or $\langle r, e_2\rangle \in G_\sigma.\mathtt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathsf{Evts}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. By Lemma 5, G_ρ is executable. Since $G_\rho.\mathsf{Evts} \subseteq G_\sigma.\mathsf{Evts}$ and $G_\sigma.\mathtt{rf}|_{S_\sigma} \subseteq G_\sigma.\mathtt{hb}$, G_ρ is reads-from feasible. Since $G_\sigma.(\mathsf{po} \cup \mathtt{sync}) \subseteq G_\sigma.\mathtt{hb}$, by Lemma 6, G_ρ is lock feasible.

▶ *Inserting Memory Orders.* Finally, we show that there exists a memory insertion scheme such that G_ρ is sequentially consistent. Observe that for all $\mathsf{acq}(l) \in G_\rho.\mathsf{Acqs}$, where $l \in \mathsf{LocksHeld}(e_1)$, we have $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_1$ because either $\langle \mathsf{acq}(l), e_1\rangle \in G_\sigma.\mathtt{hb}$ or $\langle \mathsf{acq}(l), e_2\rangle \in G_\sigma.\mathtt{hb}$ by definition. In the former case, since $\mathtt{hb} \subseteq \mathtt{trace}$, we know that $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_1$. In the latter case, since $l \in \mathsf{LocksHeld}(e_1)$, we can infer that there is a hb path between e_1 and $\mathsf{acq}(l)$. Then $\mathsf{acq}(l)$ has to be trace ordered before e_1 as the alternative would imply $e_1 \xrightarrow{\mathtt{hb}} \mathsf{acq}(l) \xrightarrow{\mathtt{hb}} e_2$, i.e., a contradiction with the assumption that $e_1 \parallel_{\texttt{hb}} e_2$. Then we can apply Lemma 7 and conclude that there exists a memory insertion scheme such that G_ρ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

4.6.5 Schedulable Happens-before (SHB)

Schedulable Happens-before (SHB) [MKV18] is an extension of Happens-before (HB) in which the race reported by the SHB algorithm does not have to be the first race to be sound. Like HB, it is also a partial order based algorithm that builds an shb order, which is defined as the following.

$$\texttt{shb} = (\mathsf{po} \cup \texttt{sync} \cup \texttt{rf})^+$$

The soundness theorem is stated as the following.

▶ Theorem 8
If $e_1 \parallel_{\texttt{shb}} e_2$, then $\langle e_1, e_2 \rangle$ is a sound race.

Proof. Let S_σ be a set defined as the following.

$$S_\sigma = \{e \in G_\sigma.\mathsf{Evts} \mid \langle e, e_1 \rangle \in G_\sigma.\texttt{shb} \ \vee \ \langle e, e_2 \rangle \in G_\sigma.\texttt{shb}\}$$
$$C_\sigma = G_\sigma.(\mathsf{Rds} \cup \mathsf{Wrts}) \cap S_\sigma$$

Because shb is transitive, S_σ is downward-closed w.r.t. shb. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evts} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ is the identity function. The rest of the proof follows the exact same reasoning process of the soundness proof for HB because $\texttt{hb} \subseteq \texttt{shb}$. □

4.6.6 SyncP

SyncP [MPV21] is a data race prediction algorithm that computes the event set of a potential witness and determine if $\langle e_1, e_2 \rangle$ is a data race by checking whether they are included in the set. Formally, the sync-reversal-free closure of (e_1, e_2), $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$, is a set of events defined inductively by the following rules:

- $\{\mathsf{prev}_\sigma(e_1), \mathsf{prev}_\sigma(e_2)\} \subseteq \mathsf{SRFIdeal}_\sigma(e_1, e_2)$, where $\langle \mathsf{prev}_\sigma(e), e \rangle \in G_\sigma.\mathsf{po}|_{imm}$ for all $e \in G_\sigma.\mathsf{Evts}$,
- for each event $e \in G_\sigma.\mathsf{Evts}$, if $\langle e, e' \rangle \in G_\sigma.(\mathsf{po} \cup \mathtt{rf})$ for some event $e' \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$, then $e \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$,
- for any two acquire events $\mathsf{acq}_1(l)$ and $\mathsf{acq}_2(l) \in G_\sigma.\mathsf{Acqs}$, if $\mathsf{acq}_1(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and $\mathsf{acq}_2(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and $\langle \mathsf{acq}_1(l), \mathsf{acq}_2(l) \rangle \in G_\sigma.\mathtt{trace}$, then there is a release event $\mathsf{rel}_1(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ such that $\mathsf{match}(\mathsf{rel}_1(l)) = \mathsf{acq}_1(l)$.

The soundness theorem is stated as the following.

▶ Theorem 9
If $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \varnothing$, then $\langle e_1, e_2 \rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \mathsf{SRFIdeal}_\sigma(e_1, e_2)$$
$$C_\sigma = G_\sigma.(\mathsf{Rds} \cup \mathsf{Wrts}) \cap S_\sigma$$

By the second condition of the definition of $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$, we know that S_σ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evts} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ is the identity function.

We first show that $G_\rho \rhd b$ where $b = e_1 e_2$. To start with, $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \varnothing$ comes from the assumption. In addition, No Skipping is satisfied by the base condition of the definition, i.e., $\{\mathsf{prev}(e_1), \mathsf{prev}(e_2)\} \subseteq \mathsf{SRFIdeal}_\sigma(e_1, e_2)$. Since δ is the identity function, for each $r \in G_\sigma.\mathsf{Rds}$ such that $\langle r, e_1\rangle \in G_\sigma.\mathtt{ctrl}$ or $\langle r, e_2\rangle \in G_\sigma.\mathtt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathsf{Evts}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. Since $G_\rho.\mathsf{Evts}$ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$, by Lemma 5, G_ρ is read feasible and executable. If there are two open critical sections co-exist in G_ρ, then there exists two acquire events $\mathsf{acq}_1(l)$ and $\mathsf{acq}_2(l)$ such that both their matching release events are not included in $G_\rho.\mathsf{Evts}$. However, this contradict with the third condition of the definition of $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$. Hence, for each lock $l \in \mathsf{Lock}$, there is at most one open critical section present in G_ρ. That is, G_ρ is lock feasible.

▶ *Inserting Memory Orders.* Lastly, we show that there exists a memory order insertion scheme such that G_ρ is sequentially consistent. Observe that for any $\mathsf{acq}(l) \in G_\rho.\mathsf{Acqs}$ where $l \in \mathsf{LocksHeld}(e_i)$ and $i \in \{1, 2\}$, $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_i$. This is because for each $\mathsf{acq}(l)$, if $l \in \mathsf{LocksHeld}(e_i)$, then either $\mathsf{acq}(l)$ is the acquire event of the critical section of e_i, or there are acquire event $\mathsf{acq}_i(l)$ and release events $\mathsf{rel}(l)$ and $\mathsf{rel}_i(l) \in G_\sigma$ where $\mathsf{match}(\mathsf{rel}(l)) = \mathsf{acq}(l)$, $\mathsf{match}(\mathsf{rel}_i(l)) = \mathsf{acq}_i(l)$, and $\mathsf{acq}_i(l)$ is the acquire event of e_i's critical section. In the former case, it's obvious that $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_i$. In the latter case, either $\mathsf{rel}_i(l) \xrightarrow{\mathtt{sync}} \mathsf{acq}(l)$ or $\mathsf{rel}(l) \xrightarrow{\mathtt{sync}} \mathsf{acq}_i(l)$. We know that $\mathsf{rel}_i(l) \notin G_\rho.\mathsf{Evts}$ because G_ρ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$ and $e_i \notin G_\rho.\mathsf{Evts}$. Hence, if $\mathsf{acq}(l) \in G_\rho$, it must be that $\mathsf{rel}(l) \xrightarrow{\mathtt{sync}} \mathsf{acq}_i(l)$, which implies that $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_i$. By Lemma 7, there exists an order insertion scheme such that G_ρ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

4.7 Future Work

Soundness definitions may be extended to state that every reported data race indicates either the presence of a race or a predictable deadlock. Our treatment does not currently incorporate the deadlock provision covered by the weak soundness theorem, which is used in some partial-order-based data race predictive analyses [SES12, GRX19, KMV17]. While one can modify our soundness definition with a disjunction to accommodate the weak soundness theorem, precisely capturing a predictable deadlock pattern with respect to the reported data race requires further exploration.

As mentioned in Section 4.4.4, our definition for $\mathcal{M}$-consistency assumes MCA models. Integrating the semantics of lock acquire and release events into non-MCA models remains as future work.

CHAPTER 5

Correctness of Transformations under Weak Memory Models

Transformations are used in both the contexts of compilers and predictive analyses. In compilers, before the compilation schemes are applied, source programs are often transformed in order for more optimized code to be produced after compilation. In predictive analyses, executions can be transformed to reveal concurrent bugs under weak memory models using existing analyses that assume sequential consistency. As mentioned in Chapter 1, a sound compiler transformation can be seen as a function that takes a set of executions as input and produces a set of observationally equivalent executions such that any behavior that is observable after the transformation should be expected before the transformation. Interestingly, the same notion of sound transformations can be used to extend existing predictive analyses, which assume sequential consistency, to predict concurrent bugs under weak memory models.

In this chapter, we start by diving into the sound transformations under the memory model of Java Access Modes and compare them with the sound trasnformations under the memory model of C++11, which are extensively studied by [LVK17]. We then focus our attention on sound transformations that can help existing predictive analyses, which assume sequential consistency, to catch concurrent bugs under weak memory models. In particular, in Section 5.2, we borrow two sound transformations for x86-TSO from [LV16] and show a new sound transformation for ARMv8 [ADG21].

$$\begin{tikzcd}
P_{src} \arrow[r, rightsquigarrow] \arrow[d] & P_{tgt} \arrow[d] \\
G_{src} \arrow[r, rightsquigarrow] & G_{tgt}
\end{tikzcd}$$

Figure 5.1: The relationship between programs and executions with transformations

5.1 Sound Transformations for Java Access Modes

One important aspect of compilers is the program transformations that they apply to the program. A correct compiler transformation should not introduce any new program behavior. While this is relatively simple for sequential programs, it can yield subtle issues when applying the same transformations to concurrent programs. A memory model's task is then to accommodate a set of common program transformations while still provide intuitive synchronization guarantees to the programmers. In Section 3.4 we show that Java and C/C++11 can use the same compilation scheme to Power and x86. However, Java has a stronger semantics for Volatile comparing to `seq_cst` in C/C++11 and can adopt only a strict subset of the transformations that are valid for C/C++11.

In this section, we use the set of compiler transformations detailed by [LVK17] and compare their soundness in Java with C/C++11. We provide formal proofs for the sound transformations and counter-examples for invalid transformations. We conclude this section by discussing the implications of our results.

Recall that we have informally introduced the relationships of programs and executions in the context of transformations in Chapter 2, as shown in Fig. 5.1. Let P_{src} be the program before applying a transformation and P_{tgt} is the program obtained after the transformation. G_{src} and G_{tgt} are two execution graphs assiciated with the two programs. To prove a transformation is valid, intuitively, we show that there does not exist a G_{src} of P_{src} such that it is forbidden by

Transformation		C/C++11	Java
Strengthening	[§ 5.1.1]	✓	✓
Sequentialisation	[§ 5.1.2]	✓	✓
Reordering	[§ 5.1.3]	See Fig. 5.3	
Merging	[§ 5.1.4]	See Fig. 5.4	
Register Promotion	[§ 5.1.5]	✓	For locations that does not have Volatile access

Figure 5.2: Compiler Transformations in C/C++11 and Java

JAM21 but the corresponding G_{tgt} of P_{tgt} is allowed.

▶ DEFINITION 20 (SOUND PROGRAM TRANSFORMATION)
Let P_{src} be a Java program which has a set of candidate executions, $[\![P_{src}]\!]$. Let $T : \mathcal{G} \to \mathcal{G}$ be a transformation and $G_{tgt} = T(G_{src})$ for each candidate execution G_{src} of P_{src}. Then we say T is **sound** under JAM21 if for each G_{tgt}, if G_{tgt} is JAM21-consistent, then G_{src} is also JAM21-consistent.

The results for Java comparing them C/C++11 [LVK17] are summarized in Fig. 5.2.

We explain each type of transformation shown in the table in details in the rest of this section.

5.1.1 Strengthening

Strengthening transforms the access mode of accesses to stronger access modes. It is supported by JAM21 due to the monotonicity property (see Appendix H) of the memory model. The formal theorem is the following:

▶ THEOREM 10 (STRENGTHENING)
Let G_{tgt} an execution of P_{tgt}, which is obtained from applying Strengthening to a program P_{src}. There exists an execution G_{src} of P_{src} such that:

- G_{src}.Evts = G_{tgt}.Evts
- G_{src}.po = G_{tgt}.po
- G_{src}.rf = G_{tgt}.rf

- $\forall i \in G_{src}.\mathsf{Evts}, G_{src}.AccessMode(i) \sqsubseteq G_{tgt}.AccessMode(i)$

If G_{tgt} is JAM21-consistent, then G_{src} is JAM21-consistent.

Proof. By Monotonicity of JAM21, all the constraints in G_{src} are preserved in the strengthened execution G_{tgt}. Therefore, if G_{tgt} is JAM21-consistent, so is G_{src}. $\square$

5.1.2 Sequentialization

Sequentialization transforms two concurrent accesses into accesses in a single sequential process. It is natually supported by JAM21 because sequentialization does not remove any synchronization from the program.

▶ THEOREM 11 (SEQUENTIALIZATION)
Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a sequentialization operation on a pair of accesses a and b. Let G_{tgt} be an execution of P_{tgt}. Then there exists an execution G_{src} of P_{src} such that

- $G_{src}.\mathsf{po} \cup \{\langle a, b\rangle\} = G_{tgt}.\mathsf{po}$ where $\langle a, b\rangle \notin G_{src}.\mathsf{po}$ and $\langle b, a\rangle \notin G_{src}.\mathsf{po}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{src}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$

and if G_{tgt} is JAM21-consistent, then G_{src} is JAM21-consistent.

Proof. Assume towards contradiction that G_{src} is not JAM21-consistent. Then there are two cases: either there is a $(\mathsf{po} \mid \mathtt{rf})^+$ cycle or a $\mathtt{co}$ cycle in G_{src}. Whether or not a and b are included in this cycle, adding a po edge between a and b cannot eliminate this cycle (although it might introduces new cycles). Therefore, G_{tgt} is also not JAM21-consistent, contradicting to our assumption. $\square$

5.1.3 Reordering

The operation of *reordering* can be seen as composing *deordering* with *sequentialization*. Since we know that sequentialization is sound in JAM21, we only need to show that deordering is sound in order to show reordering is sound in JAM21.

Deordering Deordering is a transformation that turns a pair of accesses related by a po relation into a pair of concurrent accesses. In effect, it removes an po edge in the execution graph.

First, we adopt the same definition of adjacent events from [LVK17]:

▶ DEFINITION 21 (ADJACENT EVENTS)
Two events a and b are **adjacent** in a partial order R if for all c, we have:

- $c \xrightarrow{\mathtt{R}} a \Rightarrow c \xrightarrow{\mathtt{R}} b$
- $b \xrightarrow{\mathtt{R}} c \Rightarrow a \xrightarrow{\mathtt{R}} c$

For Java, the table of allowed reordering two adjacent events (with each row as the first event and column as the second event) is shown in Fig. 5.3 (some of the cases are different from C11 [LVK17] and we have marked them in red). Intuitively, the sound deorderable pairs are ordered by the po edges that does not impose any synchronization in the program. Therefore, deordering (removing the po edge) does not introduce new program behavior.

To prove that JAM21 supports the reordering shown in this table, we need to prove each cell shown in the table is valid for JAM21.

▶ THEOREM 12 (DEORDERING)
Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a deordering operation on a pair of accesses a and b according to Fig. 5.3. Let G_{tgt} be an execution of P_{tgt}. Then there exists an execution G_{src} of P_{src} such that

- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\}$ where a and b are po-adjacent
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts}$

	$\texttt{R}_y^{m_2}$	$\texttt{W}_y^{m_2}$	$\texttt{RMW}_y^{m_2}$	$\texttt{F}^{m_2}$
$\texttt{R}_x^{m_1}$	$m_1 \sqsubseteq \text{Opaque}$	$m_1, m_2 \sqsubseteq \text{Opaque} \wedge (m_1 = \text{Plain} \vee m_2 = \text{Plain})$	$m_1 = \text{Plain} \wedge m_2 \sqsubseteq \text{Acquire}$	$(m_1 \sqsubseteq \text{Opaque} \wedge m_2 = \text{Release} \wedge \forall i, \texttt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin G.\text{Wrts}) \vee (m_1 = \text{Acquire} \wedge m_2 = \text{Acquire}) \vee (m_1 = \text{Acquire} \wedge m_2 = \text{Release})$
$\texttt{W}_x^{m_1}$	$m_1 \neq \text{Volatile} \vee m_2 \neq \text{Volatile}$	$m_2 \sqsubseteq \text{Opaque}$	$m_2 \sqsubseteq \text{Acquire}$	$(m_2 = \text{Acquire}) \vee (m_2 = \text{Release} \wedge \forall i, \texttt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin G.\text{Wrts}) \vee (m_2 = \text{Release} \wedge \forall i, \texttt{F}^{m_2} \xrightarrow{\text{po}} i \wedge i \in G.\text{Wrts} \Rightarrow AccessMode(i) = \text{Release})$
$\texttt{RMW}_x^{m_1}$	$m_1 \sqsubseteq \text{Release}$	$m_1 \sqsubseteq \text{Release} \wedge m_2 = \text{Plain}$	-	$(m_1 \sqsupseteq \text{Acquire} \wedge m_2 = \text{Acquire}) \vee (m_2 = \text{Release} \wedge \forall i, \texttt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow (i \in G.\text{Rds} \vee (i \in G.\text{Wrts} \wedge AccessMode(i) = \text{Release})))$
$\texttt{F}^{m_1}$	$(m_1 = \text{Release}) \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \texttt{F}^{m_1} \Rightarrow i \notin G.\text{Rds})$	$m_1 = \text{Release} \wedge m_2 \sqsupseteq \text{Release} \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \texttt{F}^{m_1} \Rightarrow i \notin G.\text{Rds})$	$m_1 = \text{Release} \wedge m_2 \sqsupseteq \text{Release} \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \texttt{F}^{m_1} \Rightarrow i \notin G.\text{Rds})$	$(m_1 = \text{Release} \wedge m_2 = \text{Acquire}) \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \texttt{F}^{m_1} \Rightarrow i \notin G.\text{Rds}) \vee (m_2 = \text{Release} \wedge \forall i, \texttt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin G.\text{Wrts})$

Figure 5.3: Allowed Deordering Pairs in JAM21

- $G_{src}.\texttt{to} = G_{tgt}.\texttt{to}$
- $G_{src}.\texttt{IW} = G_{tgt}.\texttt{IW}$
- $\forall i \in G_{src}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$

and if G_{tgt} is JAM21-consistent, then G_{src} is JAM21-consistent.

Please see Appendix T for the proof.

Reordering, as mentioned previously, can be decomposed into two steps: deordering and sequentialization. Since we have already shown the soundness of the two transformations, the soundness of reordering follows naturally.

▶ COROLLARY 2 (REORDERING)
JAM21 supports the reordering transformation for pairs of adjacent accesses shown in Fig. 5.3.

5.1.4 Merging

Merging transforms two adjacent accesses into one single equivalent access to reduce the number of memory accesses in the program. We have grouped all types of merging transformations appeared in C/C++11 [LVK17] here in one section. A summarized result of mergable pairs comparing with C/C++11 can be found in Fig. 5.4. The results are mostly similar except for Volatile. Many merging transformation are invalid for Volatile because they remove the cross-thread synchronization of Volatile.

5.1.4.1 Read-Read Merging

Read-read merging is sometimes done when the compiler is optimizing redundant loads in the same thread. When we are encountering two consecutive reads to the same location, the first read is unchanged but the second read becomes a local read without accessing the memory.

Let a' and b be two adjacent read accesses reading from the same write access a. $a \xrightarrow{\texttt{rf}} a'$ and $a \xrightarrow{\texttt{rf}} b$, and $a' \xrightarrow{\texttt{po}} b$. Assuming $AccessMode(a') = AccessMode(b)$, then

Name	C/C++11	Java
Read-read Merging	$\mathtt{R}^{m};\mathtt{R}^{m} \rightsquigarrow \mathtt{R}^{m}$	$\mathtt{R}^{m\sqsubseteq \mathsf{Acq}};\mathtt{R}^{m\sqsubseteq \mathsf{Acq}} \rightsquigarrow \mathtt{R}^{m}$
Write-write Merging	$\mathtt{W}^{m};\mathtt{W}^{m} \rightsquigarrow \mathtt{W}^{m}$	$\mathtt{W}^{m\sqsubseteq \mathsf{Rel}};\mathtt{W}^{m\sqsubseteq \mathsf{Rel}} \rightsquigarrow \mathtt{W}^{m}$
Write/RMW-read Merging	$\mathtt{W}^{m};\mathtt{R}^{\mathtt{acq}} \rightsquigarrow \mathtt{W}^{m}$	$\mathtt{W}^{m};\mathtt{R}^{m\sqsubseteq \mathsf{Opq}} \rightsquigarrow \mathtt{W}^{m}$
	$\mathtt{W}^{\mathtt{sc}};\mathtt{R}^{\mathtt{sc}} \rightsquigarrow \mathtt{W}^{\mathtt{sc}}$	✗
	$\mathtt{RMW}^{m};\mathtt{R}^{m_r\sqsubseteq m} \rightsquigarrow \mathtt{RMW}^{m}$	$\mathtt{RMW}^{m};\mathtt{R}^{m\sqsubseteq \mathsf{Opq}} \rightsquigarrow \mathtt{RMW}^{m}$
Write-RMW Merging	$\mathtt{W}^{m_w\sqsubseteq m};\mathtt{RMW}^{m} \rightsquigarrow \mathtt{W}^{m_w}$	$\mathtt{W}^{m_w\sqsubseteq \mathsf{Rel}};\mathtt{RMW}^{m\sqsubset \mathsf{Vol}} \rightsquigarrow \mathtt{W}^{m_w}$
RMW-RMW Merging	$\mathtt{RMW}^{m};\mathtt{RMW}^{m} \rightsquigarrow \mathtt{RMW}^{m}$	$\mathtt{RMW}^{m\sqsubset \mathsf{Vol}};\mathtt{RMW}^{m\sqsubset \mathsf{Vol}} \rightsquigarrow \mathtt{RMW}^{m}$
Fence-fence Merging	$\mathtt{F}^{m};\mathtt{F}^{m} \rightsquigarrow \mathtt{F}^{m}$	$\mathtt{F}^{m};\mathtt{F}^{m} \rightsquigarrow \mathtt{F}^{m}$

Figure 5.4: Mergable Pairs in C/C++11 [LVK17] and Java

- $\forall i, a' \xrightarrow{\mathtt{po}} i \Rightarrow b \xrightarrow{\mathtt{po}} i$
- $\forall i, a' \xrightarrow{\mathtt{ra}} i \Rightarrow b \xrightarrow{\mathtt{ra}} i$
- $\forall i, a' \xrightarrow{\mathtt{push}} i \Rightarrow b \xrightarrow{\mathtt{push}} i$
- $\forall j, j \xrightarrow{\mathtt{po}} b \Rightarrow j \xrightarrow{\mathtt{po}} a'$

For executions, this corresponds to the following transformation in the execution graph: since the value of $r1$ and $r2$ are guaranteed to have the same value in P_{tgt}, we know that this corresponds to the execution of P_{src} where the two read accesses read from the same write access. Then we want to show that, if G_{tgt} is JAM21-consistent, G_{src} is also JAM21-consistent.

▶ THEOREM 13 (READ-READ MERGING)
Let G_{tgt} be an JAM21-consistent execution. Let $a \in G_{tgt}.\mathsf{Rds}\backslash\mathtt{RMW}$ and let $a' \in G_{tgt}.\mathsf{Evts}$ such that $a \xrightarrow{\mathtt{rf}} a'$. Let $b \notin G_{tgt}.\mathsf{Evts}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{po}} j\}$

- $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf} \cup \{\langle a', b\rangle\}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $G_{src}.\texttt{to} = G_{tgt}.\texttt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\texttt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\texttt{to}} j\}$
- $G_{src}.\texttt{IW} = G_{tgt}.\texttt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $b \in G_{src}.\mathsf{Rds}$
- $G_{src}.AccessMode(b) = G_{src}.AccessMode(a) \sqsubseteq \mathsf{Acquire}$

and G_{src} is JAM21-consistent.

Please see Appendix T for the proof.

Note that JAM21 does not allow read-read merging if the two read accesses are both Volatile mode reads. We provide an example of this in Appendix T.

5.1.4.2 Write-Write Merging

The write-write merge transformation refers to the program transformation that merges two consecutive write operations into one by removing the former one. JAM21 support write-write merge when the access modes of the two writes are the same and they are not Volatile mode accesses.

Let a and b be the two adjacent writes such that $a \xrightarrow{\texttt{po}} b$. We once again have the properties:

- $\forall i, i \xrightarrow{\texttt{po}} a \Rightarrow i \xrightarrow{\texttt{po}} b$
- $\forall j, b \xrightarrow{\texttt{po}} j \Rightarrow a \xrightarrow{\texttt{po}} j$
- $\forall i, i \xrightarrow{\texttt{ra}} a \Rightarrow i \xrightarrow{\texttt{ra}} b$

We have the following theorem.

▶ THEOREM 14 (WRITE-WRITE MERGING)
Let G_{tgt} be an JAM21-consistent execution. Let $b \in G_{tgt}.\mathsf{Wrts} \setminus \mathtt{RMW}$ and let $a \notin G_{tgt}.\mathsf{Evts}$ and $loc(a) = loc(b) \land \forall i \in G_{tgt}.\mathsf{Wrts}, loc(i) = loc(b) \Rightarrow val(a) \neq val(i)$. There exists a G_{src} such that:

- $G_{src}.\mathtt{po} = G_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{a\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $a \in G_{src}.\mathsf{Wrts}$
- $G_{src}.AccessMode(a) = G_{src}.AccessMode(b) \sqsubseteq \mathsf{Release}$

and G_{src} is JAM21-consistent.

Please see Appendix T for the proof.

Note that write-write merging is not valid for Volatile mode writes. We provide an example of this in Appendix T.

5.1.4.3 Write/RMW-read Merging

The Write/RMW-read merging refers to the program transformation that merges a write/RMW and a read into a single write/RMW and a local access.

Similarly, the transformation with an RMW operation and a read operation optimizes the latter read operation to read locally and in effect removes a memory load operation in the execution graph.

JAM21 only support this transformation when the read operation is (or is weaker than) Opaque mode which is different from RC11 [LVK17]'s result for C/C++11. We provide a counter-example in Appendix T to show that write/RMW-read merging is invalid when the read is (or is stronger than) Acquire mode.

▶ THEOREM 15 (WRITE/RMW-READ MERGING)
Let G_{tgt} be a JAM21-consistent execution. Let $a \in G_{tgt}.\mathsf{Wrts}$ and $b \notin G_{tgt}.\mathsf{Evts}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $b \in G_{src}.\mathsf{Rds}$
- $G_{src}.loc(b) = G_{src}.loc(a)$
- $G_{src}.val(b) = G_{src}.val(a)$
- $G_{src}.\mathtt{po} = G_{tgt}.\mathtt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a,b\rangle\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.AccessMode(b) \sqsubseteq \mathsf{Opaque}$

Please see Appendix T for the proof.

5.1.4.4 Write-RMW Merging

The write-RMW merging refers to the program transformation that merges a write and a consecutive RMW operation into a write with the value of the RMW. For example, if we have the following pattern in a program:

```
x = 1;
x.getAndSet(1,2);
```

It can be tranformed to:

```
x = 2;
```

Similar to write-write merging, Jam21 supports write-RMW merging when the access mode of the write is {Opaque, Release} and the access mode of the RMW is {Acquire, Release}.

▶ Theorem 16 (Write-RMW Merging)
Let G_{tgt} be a Jam21-consistent execution. Let $b \in G_{tgt}.\mathsf{Wrts} \setminus G_{tgt}.\mathtt{RMW}$, $a \notin G_{tgt}.\mathsf{Evts}$ and $v \in \mathtt{Val}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{a\}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.AccessMode(a) \in \{\text{Opaque, Release}\}$
- $G_{src}.AccessMode(b) \in \{\text{Acquire, Release}\}$
- $G_{src}.loc(b) = G_{src}.loc(a)$
- $b \in G_{src}.\mathtt{RMW}$
- $G_{src}.val(b) = (G_{src}.val(a), v)$
- $G_{src}.\mathtt{po} = G_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$

and G_{src} is Jam21-consistent.

Please see Appendix T for the proof.

5.1.4.5 RMW-RMW Merging

The RMW-RMW merging transformation refers to the program transformation that merges two consecutive RMW operations into one such that it has the first RMW's (expected) read value and the second RMW's write value. For example, if we have the following pattern in a program:

```
x.getandSet(1,2);
x.getandSet(2,3);
```

then it might be transformed into:

```
x.getAndSet(1,3);
```

The RMW-RMW merging transformation is essentially the same as write-write merging and read-read merging described previously. Therefore, the set of constraints on valid access modes for merging is the intersection of the two. That is, two RMWs are mergeable if they are both Acquire mode or Release mode. For the counter-examples showing this transformation is invalid for Volatile accesses, please see the examples for write-write and read-read merging.

▶ Theorem 17 (RMW-RMW Merging)
Let G_{tgt} be a Jam21-consistent execution. Let x be a memory location and $a \in G_{tgt}.\mathsf{Evts}$ with $G_{tgt}.val(a) = (v_r, v_w)$, $G_{tgt}.loc(a) = x$, and $G_{tgt}.AccessMode(a) \in \{\mathsf{Release}, \mathsf{Acquire}\}$. Let $b \notin G_{tgt}.\mathsf{Evts}$, there exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.val(a) = (v_r, v)$
- $G_{src}.val(b) = (v, v_w)$
- $G_{src}.loc(b) = x$
- $G_{src}.AccessMode(b) = G_{src}.AccessMode(a) \in \{\mathsf{Release}, \mathsf{Acquire}\}$
- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathsf{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathsf{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{to}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$

and G_{src} is Jam21-consistent.

Please see Appendix T for the proof.

5.1.4.6 Fence-fence Merging

The Fence-fence merging refers to the program transformation that merges two consecutive fences of the same access mode into one. For example, if we have:

```
VarHandle.fullFence();
VarHandle.fullFence();
```

then it can be optimized to:

```
VarHandle.fullFence();
```

Since Jam21 is fence-based such that each fence is converted into an edge between memory accesses, this is trivially supported since the execution graph before and after the transformation is exactly the same.

5.1.5 Register Promotion for Non-shared Variable

Register Promotion promotes memory accesses of a non-shared memory location to local registers. It has the effect of removing memory accesses for thread-local variables. Jam21 only supports register promotion for variables without any Volatile accesses in the program. For non-Volatile accesses, since the variable is not shared across threads, it is safe to remove them without worrying about removing synchronization from the program. In contrast, Volatile accesses impose cross-thread synchronizations with Volatile accesses for other variables, so removing such accesses can potentially remove important synchronization in the program and introduce new behaviors that were previously forbidden by the memory model. We provide a counter-example in this section showing that we cannot promote Volatile accesses to local register accesses even if the location is only accessed by one thread.

Suppose all accesses to a memory location are in the same thread, the transformation can be seen as two steps:

1. Weakening the accesses to Plain mode accesses
2. Removing the Plain mode accesses

▶ Theorem 18 (Weakening for non-shared variable)
Let G_{tgt} be a Jam21-consistent execution such that, for all accesses i and j in G_{tgt}.Evts, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in G_{tgt}$.Evts, $loc(i) = x \Rightarrow AccessMode(i) =$ Plain. There exists an execution G_{src} such that:

- G_{src}.Evts = G_{tgt}.Evts
- G_{src}.`po` = G_{tgt}.`po`
- G_{src}.`rf` = G_{tgt}.`rf`
- G_{src}.`to` = G_{tgt}.`to`
- G_{src}.`IW` = G_{tgt}.`IW`
- $\forall i \in G_{src}$.Evts, $loc(i) = x \Rightarrow AccessMode(i) \in$ {Release, Acquire}

and G_{src} is Jam21-consistent.

Please see Appendix T for the proof.

▶ Theorem 19 (Removing Plain accesses for non-shared variable)
Let G_{tgt} be a Jam21-consistent execution. Let x be a memory location and for all $i \in G_{tgt}$.Evts such that $loc(i) = x$, $Tid(i) = t$ for some x and t. Let $a \notin G_{tgt}$.Evts. There is a G_{src} such that:

- G_{src}.Evts $= G_{tgt}$.Evts $\cup \{a\}$
- $G_{src}.loc(a) = x$
- $G_{src}.AccessMode(a) =$ Plain
- G_{src}.`po` $\supset G_{tgt}$.`po`
- for all $i \in G_{src}$.Evts such that $G_{src}.loc(i) = x$, $i \xrightarrow{\text{po}} a$ or $a \xrightarrow{\text{po}} i$
- G_{src}.`rf` $= G_{tgt}$.`rf` if $a \in G_{src}$.Wrts\\`RMW`, otherwise, G_{src}.`rf` $= G_{tgt}$.`rf` $\cup \{\langle i, a\rangle\}$ such that $(i \in G_{src}$.Wrts$) \wedge (loc(i) = x) \wedge (i \xrightarrow{\text{po}} a) \wedge (\forall j \in G_{src}$.Evts, $(loc(j) = x) \wedge (j \xrightarrow{\text{po}} a) \Rightarrow (j \xrightarrow{\text{po}} i))$.
- G_{src}.`to` = G_{tgt}.`to`
- G_{src}.`IW` = G_{tgt}.`IW`

and G_{src} is Jam21-consistent.

Please see Appendix T for the proof.

Counter Example We now show a counter example for invalid register promotion on locations with Volatile accesses. Consider the following program:

```
Thread0 {
  int r1 = X.getOpaque(); // 1
  int r2 = X.getOpaque(); // 2
}

Thread1 {
  int r3 = Y.getOpaque(); // 1
  int r4 = Y.getOpaque(); // 2
}
```

```
Thread2 {
  X.setOpaque(2);
  Z.setVolatile(1);
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

An execution with the annotated values in this program is not allowed by JAM21. The execution graph before the transformation is shown in Fig. 5.5. First note that the Volatile access on z also has Release semantics due to the monotonicity of access modes, which yields the `ra` edge in Thread 2. The total order among `push` edges gives use two cases:

1. $\texttt{Wz = 1} \xrightarrow{\texttt{vvo}} \texttt{Wx = 1}$. Since $\texttt{Wx = 2} \xrightarrow{\texttt{ra}} \texttt{Wz = 1}$ and $\texttt{ra} \subseteq \texttt{vvo}$ and $\texttt{vvo}^+ \subseteq \texttt{vo}$, we have $\texttt{Wx = 2} \xrightarrow{\texttt{vo}} \texttt{Wx = 1}$, which contradict with the `co` edge established by the observation from Thread 0.

2. $\texttt{Wy = 2} \xrightarrow{\texttt{vvo}} \texttt{Wy = 1}$. This contradict with the `co` edge established by the observation from Thread 1.

In both cases there is a contradiction (a `co` cycle). Therefore, this execution is forbidden by JAM21.

In this example, the memory location z is only accessed by Thread 2. It maybe tempting to promote z to a local register on Thread 2 to reduce the number of memory instructions, which yields the following program:

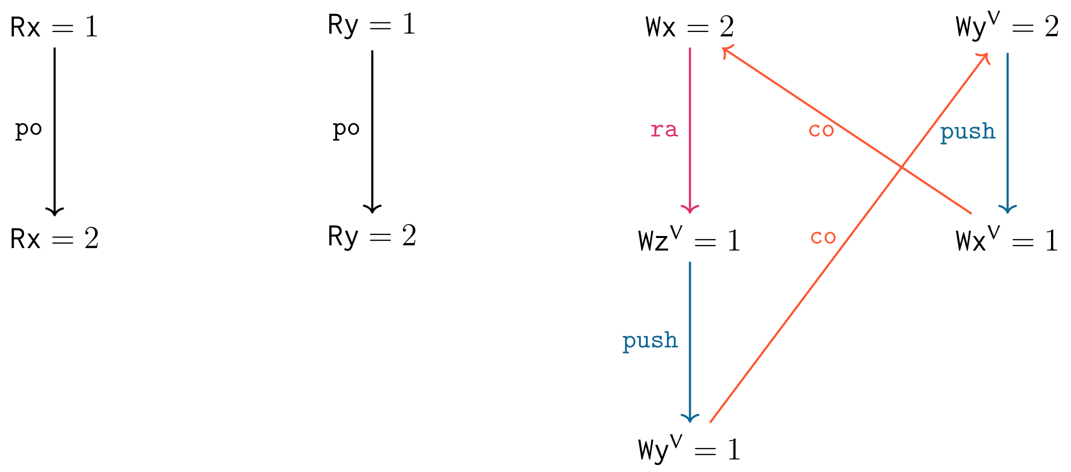


Figure 5.5: Before Register Promotion on Volatile access (Forbidden)

```
Thread0 {
  int r1 = X.getOpaque();
  int r2 = X.getOpaque();
}

Thread1 {
  int r3 = Y.getOpaque();
  int r4 = Y.getOpaque();
}
```

```
Thread2 {
  X.setOpaque(2);
  int z = 1
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

The execution graph after the transformation is shown in Fig. 5.6.

The annotated program behavior becomes allowed by JAM21 after the transformation. As the execution graph shows, since Volatile accesses also have cross-thread synchronization effect, we cannot simply weaken it to a Plain access without introducing new program behaviors.

5.1.6 Why are many transformations invalid for Volatile?

As we have shown, many local transformations are invalid for Volatile accesses under JAM21. This is not a surprise and is intended to provide programmers a more intuitive semantics for Volatile

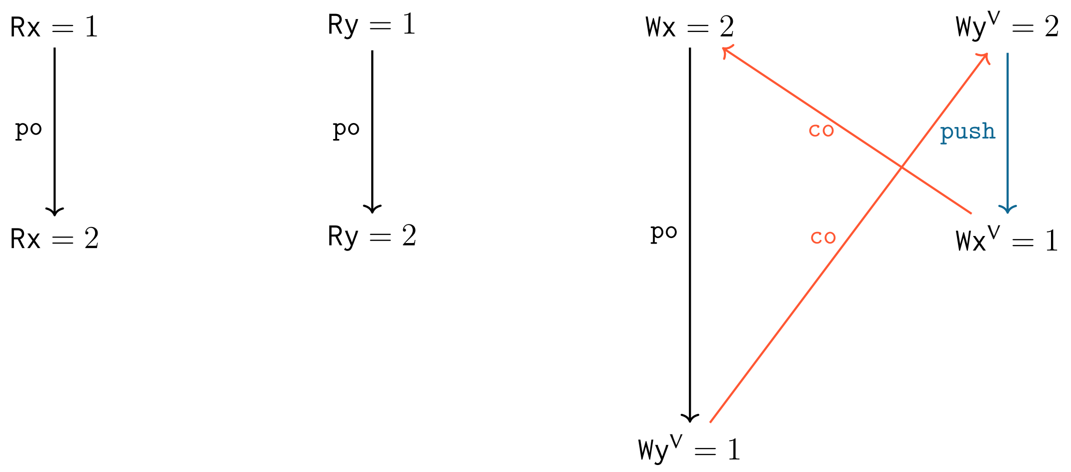


Figure 5.6: After Register Promotion on Volatile access (Allowed)

accesses.

First, as we have confirmed with the author of [Lea18], Java's Access Modes intend equivalent semantics for Volatile mode and `fullFence()`. In this way, the programmers can easily understand the semantics of both once they understand `fullFence()`. To accurately capture this intention, JAM21 used a fence-based approach with push order to model Volatile mode. As we described in Section 3.2, `fullFence()` in Java has cross-thread synchronization effects. As a result, any local program transformation that removes a Volatile access from the execution graph may also remove its cross-thread synchronization, and might introduce new program behavior after the transformation. Therefore, those transformations on Volatile accesses are mostly not allowed by JAM21. On the other hand, the sc fence in C/C++11 [LVK17] has slightly stronger synchronization effect than sc accesses so that they can be used to restore sequential consistency when inserted between every pair of accesses. Some of the transformations are allowed to apply to sc accesses but not to the fence version of the program.

In addition, restricting the set of possible transformations that is allowed to apply to Volatile variables can keep the coding process simple for programmers. From the programmers' perspective, one of the biggest challenges of developing and debugging concurrent programs comes from the compiler transformations that introduces surprising program behaviors that are not observable under sequential consistency. Therefore, restricting the set of possible transformations on Volatile accesses can restrict the set of surprising program behaviors that can happen when using Volatile mode, making the development process simpler. From this perspective, JAM21 provides more synchronization guarantees for Volatile mode than C/C++11 for sc mode atomic accesses.

Lastly, as we have confirmed with the author of [Lea18] the current implementation of OpenJDK JVM does not apply those transformations on Volatile accesses.

5.2 Sound Transformations for Predictive Analyses

In the previous section, we have seen a set of sound transformations for a weak memory model JAM21. Just like JAM21, there are sound transformations for other weak memory models as well, such as x86-TSO and ARMv8. Among all the sound transformations, a subset of the transformation can transform an weak-memory consistent execution graph into a sequentially consistent one. In this case, since the execution graphs before and after the transformation represents the same program behavior, we say the behavior of the weak-memory consistent execution is *explained* by the sequentially consistent execution [LV16].

This feature of explaining a weak memory behavior using a sequentially consistent execution is especially helpful for predictive analyses, since many existing predictive analyses are *trace*-based, i.e., they expect the input execution is a sequential trace and predict bugs that correspond to sequential witness traces. Although sequentially consistent executions can be linearized to sequential traces, weak memory consistent execution graphs are not linearizable to traces due to the cycles that they allow. As a result, existing analysis cannot be directly used to predict bugs under weak memory models. However, if we augment the analyses with transformations such that each of the weak memory bug witness can be explained by a sequentially consistent execution, then the existing trace-based algorithms can still be applied and those sequentially consistent executions are sufficient for showing the soundness of the results.

In this section, we borrow two of such transformations from [LV16] for explaining x86-TSO and define a new transformation that explains ARMv8. We then demonstrate how a new data race can be catched by adding transformation to an existing data race predictive analysis.

5.2.1 Sound Transformation for x86-TSO

The same trace-based approaches from existing data race prediction works cannot be directly applied to executions under x86-TSO. The problem is that only sequentially consistent execution graphs has a set of well-defined execution traces, but not TSO-consistent graphs. This is essentially because the execution cycles that are allowed under TSO are not linearizable. Therefore, we need a transformation function from a TSO-consistent execution graph to an sequentially consistent execution graph so that we can find a trace representing the TSO-consistent execution graph. The transformation functions are well-studied by [LV16].

The main aim of [LV16] was to give a more intuitive explanation of relaxed memory models using the same technique for reasoning about sequentially consistent programs. This was achieved by transforming relaxed memory execution graphs to sequentially consistent execution graphs. While there were several relaxed memory models cannot be explained via this approach, all TSO-consistent execution graphs can be transformed to sequentially consistent execution graphs via two types of program transformations. We provide the definitions of the two transformations below.

▶ Definition 22 (Write-Read Reordering [LV16])
For an execution graph G and event a and b, ReorderWR(a, b) is the execution G' obtained from G by inverting the program order from a to b, i.e., it is given by: $G'.\mathtt{po} = (G.\mathtt{po} \setminus \{\langle a, b\rangle\}) \cup \{\langle b, a\rangle\}$, and $G'.\mathtt{C} = G.\mathtt{C}$ for every other component $\mathtt{C}$. ReorderWR(a, b) is defined only when $\langle a, b\rangle \in [\mathtt{W}]; \mathtt{po}|_{imm}; [\mathtt{R}]$ and $a.loc \neq b.loc$.

▶ DEFINITION 23 (READ ELIMINATION [LV16])
For an execution G and events a and b, RemoveWR(a, b) is the execution graph G' obtained by removing b from G, i.e., G' is given by: $G'.\mathsf{Evts} = G.\mathsf{Evts} \setminus \{b\}$, and $G'.\mathtt{C} = G.\mathtt{C} \cap (G'.\mathsf{Evts} \times G'.\mathsf{Evts})$ for every other component $\mathtt{C}$. RemoveWR(a, b) is defined only when $\langle a, b\rangle \in [\mathtt{W}]; \mathtt{po}|_{imm}; [\mathtt{R}]$, $a.loc = b.loc$, and $a.val = b.val$.

We denote $G \rightsquigarrow G'$ is G' is obtained by applying either of the transformations on G.

It's well-known that the two transformations are both sound [LV16].

▶ PROPOSITION 6 (SOUNDNESS OF TRANSFORMATIONS UNDER TSO [LV16])
If $G \rightsquigarrow G'$ and G' is TSO-consistent, then so is G.

Note that a transformation is still sound if G is consistent but G' is inconsistent under TSO. In general, as long as a transformation does not introduce any new behavior, i.e., transforming an inconsistent execution into a consistent one under the same memory model, it is considered sound. Therefore, the above proposition states the consistency implication in the opposite direction of the transformation.

5.2.2 Sound Transformation for ARMv8

In this section, we define a reordering transformation for ARMv8 and prove its soundness. We use the ARMv8 model for fixed-size integer accesses from [ADG21] to show the soundness of the transformation.

▶ DEFINITION 24
Let G be a plain execution graph, $a, b \in G.\mathsf{Evts}$ with $\langle a, b\rangle \in G.\mathtt{po}|_{imm} \setminus (G.\mathtt{addr} \cup G.\mathtt{ctrl} \cup G.\mathtt{data})$ and $loc(a) \neq loc(b)$. Then $G' = \mathsf{Reorder}(G, a, b)$ is defined as:

- if there is $e \in G.\mathsf{Evts}$ such that $\langle e, a\rangle \in G.\mathtt{addr}$ and b is a write, then $G'.\mathtt{po} = G.\mathtt{po} \setminus \{\langle a, b\rangle\} \cup \{\langle b, a\rangle\}$. $G'.\mathtt{addr} = G.\mathtt{addr} \cup \{\langle e, b\rangle\}$. $G'.\mathtt{C} = G.\mathtt{C}$ for other components $\mathtt{C}$.
- otherwise, $G'.\mathtt{po} = G.\mathtt{po} \setminus \{\langle a, b\rangle\} \cup \{\langle b, a\rangle\}$. $G'.\mathtt{C} = G.\mathtt{C}$ for other components $\mathtt{C}$.

The purpose of distinguish the two cases is to preserve all the derived dob orders from the original execution graph so that the transformation stays sound. Here we are inserting an *artificial* address dependency to enforce the order. Translating to program transformation, this corresponds to the insertion of an `XOR` instruction on the destination register of e with itself (note that e is a read event since there is already an address dependency from e to a) and add the result of the `XOR` (which is always 0) to the address used in b. For more details of this transformation, please see [MSS12].

▶ Lemma 9
Let $G' = \mathsf{Reorder}(G, a, b)$. Then $G.\mathtt{dob} \subseteq G'.\mathtt{dob}$.

Proof. We show that for each $\langle e_1, e_2\rangle \in G.\mathtt{dob}$, $\langle e_1, e_2\rangle \in G'.\mathtt{dob}$. We analyze each case of `dob` separately.

- $\langle e_1, e_2\rangle \in G.\mathtt{addr} \cup G.\mathtt{data} \cup (G.\mathtt{ctrl}; [G.\mathsf{Wrts}])$. As the transformation does not change any existing component `C` in G other than $\mathsf{po}|_{imm}$, the lemma holds in this case.

- $\langle e_1, e_2\rangle \in (G.\mathtt{addr}; G.\mathsf{po}; [G.\mathsf{Wrts}])$. Then it means there is e_3 such that $\langle e_1, e_3\rangle \in G.\mathtt{addr}$, $\langle e_3, e_2\rangle \in G.\mathsf{po}$ and $e_2 \in G.\mathsf{Wrts}$. If $\langle e_3, e_2\rangle \in G.\mathsf{po}|_{imm} \backslash (G.\mathtt{addr} \cup G.\mathtt{ctrl} \cup G.\mathtt{data})$ and $loc(e_2) \neq loc(e_3)$, then the first case in Definition 24 applies and $\langle e_1, e_2\rangle \in G'.\mathtt{addr} \subseteq G'.\mathtt{dob}$. Otherwise, the relations are not changed in G' and $\langle e_1, e_2\rangle \in (G'.\mathtt{addr}; G'.\mathsf{po}; [G'.\mathsf{Wrts}]) \subseteq G'.\mathtt{dob}$.

- $\langle e_1, e_2\rangle \in ((G.\mathtt{addr} \cup G.\mathtt{data}); G.\mathtt{lrs})$. Then there is $e_3 \in G.\mathsf{Evts}$ such that $\langle e_1, e_3\rangle \in G.\mathtt{addr}$ and $\langle e_3, e_2\rangle \in G.\mathtt{lrs} \subseteq G.\mathtt{po\text{-}loc}$. $G.\mathtt{data}$, $G.\mathtt{addr}$ and $G.\mathtt{po\text{-}loc}$ are preserved by the transformation. That is, $G.\mathtt{data} = G'.\mathtt{data}$, $G.\mathtt{addr} \subseteq G'.\mathtt{addr}$ and $G.\mathtt{po\text{-}loc} = G'.\mathtt{po\text{-}loc}$. Therefore, $\langle e_1, e_2\rangle \in (G.\mathtt{addr}; G.\mathtt{po\text{-}loc})$. In addition, there cannot be any intervening write with the same location as e_2 being reordered before e_2 since the transformation requires the locations of the two events to be different. Therefore, $\langle e_3, e_2\rangle \in G'.\mathtt{lrs}$.

□

Note that the converse is not true. After the transformation, there may be more derived dob orders in the execution graph. However, this does *not* affect the soundness of the transformation as it monotonically adds more ordering constraints to the execution graph. Therefore, it follows that this transformation is sound.

▶ Lemma 10
Let G be a plain execution and $G' = \text{Reorder}(G, a, b)$ for some events $a, b \in G.\text{Evts}$. If G' is ARMv8-consistent, then so is G.

Proof. As G' is ARMv8-consistent, we know that it satisfies the two requirements: the Internal Visibility requirement and the External Visibility requirement. For Internal Visibility requirement, we have $G'.\texttt{po-loc} \cup G'.\texttt{com}$ is acyclic. As the transformation does not change po-loc and com, we get that $G.\texttt{po-loc} \cup G.\texttt{com}$ is also acyclic, hence satisfying the Internal Visibility requirement. For External Visibility requirement, by Lemma 9, we get $G.\texttt{dob} \subseteq G'.\texttt{dob}$. In addition, $G.\texttt{po-loc} = G'.\texttt{po-loc}$ and $G.\texttt{com} = G'.\texttt{com}$. Therefore, $G.(\texttt{com} \cup \texttt{po-loc} \cup \texttt{dob}) \subseteq G'.(\texttt{com} \cup \texttt{po-loc} \cup \texttt{dob})$. Hence, we get $G.((\texttt{rfe} \cup \texttt{fre} \cup \texttt{coe}) \cup (\texttt{po-loc}; [\text{Wrts}]) \cup \texttt{dob}) \subseteq G'.(\texttt{rfe} \cup \texttt{fre} \cup \texttt{coe} \cup (\texttt{po-loc}; [\text{Wrts}]) \cup \texttt{dob})$. By induction, the same relation holds for their transitive closures. Thus, $G.\texttt{ob} \subseteq G'.\texttt{ob}$. Since $G'.\texttt{ob}$ is irreflexive, so is $G.\texttt{ob}$. □

Let G be a plain execution graph and we denote $G \rightsquigarrow G'$ if G' is obtained by applying a sequence of the Reorder$(.)$ transformations starting from G. By induction on the sequence along with Lemma 10, we can see that applying a sequence of the Reorder$(.)$ transformation is sound.

▶ Corollary 3
Let G be a plain execution and G' is an execution graph obtained by applying a sequence of Reorder$(.)$ on G, i.e., $G \rightsquigarrow G'$. If G' is ARMv8-consistent, then so is G.

5.2.3 Using Sound Transformations in Predictive Analyses

In this section, we augment an existing predictive data race analysis, M2 [Pav20], with transformations to predict data races under x86-TSO.

5.2.3.1 A Motivating Example

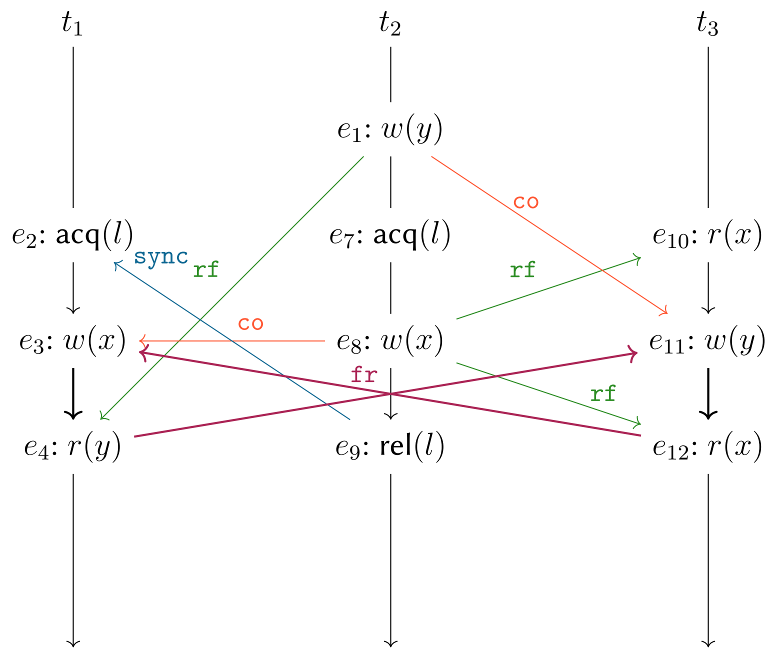
In this section, we provide an example of a TSO data race. The example was originally from the paper by [Pav20, Figure 8] to show a non-SC-race.

The input execution traces are represented in the form shown in Fig. 5.7a. Each column represents a thread and each row represents a time-stamp. At each time-stamp, there is exactly one event being executed. Events from different threads can be executed in a interleaving style while respecting the mutual exclusion property of locks. We use e_i to identify each event, where i is the time-stamp that e_i is executed. There are four types of events: read, write, lock acquire, and lock release. We write $r(x)$ and $w(x)$ for a read and a write event on a memory location x, and $\mathsf{acq}(l)$ and $\mathsf{rel}(l)$ for an acquire and a release event on a lock l. The highlighted events are reported as data races. In Fig. 5.7a, the two highlighted events, e_5 and e_{13} is not a data race under sequential consistency. Indeed,

- If e_5 and e_{13} is a data race, then Fig. 5.7b shows all the events that have to occur before e_5 and e_{13}. The order among these events has to respect the program order. Therefore, the order of events on each thread follows the same order as captured in the input trace from 5.7a.
- M2 requires each read events to read from the same write event as they appeared in the input trace to maintain soundness. Therefore, we have $e_8 \to e_{10}$ and $e_8 \to e_{12}$ for location x, and $e_1 \to e_4$ for location y.
- Since e_5 is in a critical section protected by lock l, its critical section has to be ordered after all other critical sections protected by the same lock. In this example, it means the critical section on t_2 has to be ordered before the critical section on t_1. Hence, we can infer $e_9 \to e_2$.
- By the transitivity of the partial order constructed so far, we can see that $e_8 \to e_3$. Since there are two read events on t_3 reading from e_8, then they must be ordered before e_3. Therefore, $e_{12} \to e_3$.

	t_1	t_2	t_3
1		$w(y)$	
2	acq(l)		
3	$w(x)$		
4	$r(y)$		
5	**w(z)**		
6	rel(l)		
7		acq(l)	
8		$w(x)$	
9		rel(l)	
10			$r(x)$
11			$w(y)$
12			$r(x)$
13			**r(z)**

(a) The Input Trace



(b) A TSO-Consistent Witness Execution

Figure 5.7: Example from Fig. 8 in [Pav20]

- Since $e_8 \rightarrow e_{10}$, by the transitivity of the partial order again, we have $e_1 \rightarrow e_{11}$. Since e_4 reads from e_1, then it must be that $e_4 \rightarrow e_{11}$.

- But now a cycle occurs: $e_3 \rightarrow e_4 \rightarrow e_{11} \rightarrow e_{12} \rightarrow e_3$.

On the other hand, the cycle derived in Fig. 5.7b is allowed under TSO [OSS09]. Indeed, Fig. 5.7b also shows the execution with orders among events augmented with specific semantics defined by the axiom memory model of TSO. The po order $e_3 \rightarrow e_4$ and $e_{11} \rightarrow e_{12}$ are not preserved program order. As a result, the $\mathsf{po} \cup \mathtt{fr}$ cycle shown in the figure is allowed by the TSO model. Therefore, we can conclude that e_5 and e_{13} form a data race under TSO, which M2 is not able to directly predict.

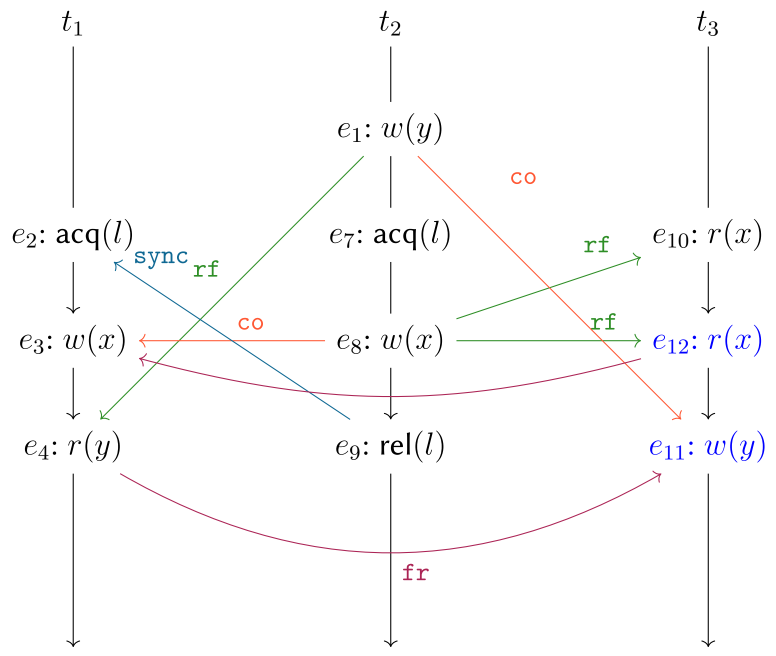
From Section 5.2.1, we have seen that there are two sound transformations to explain TSO behaviors under sequential consistency. We can use one of these two transformations and let M2 predict a witness trace that explain the behavior in Fig. 5.7b. As shown in Fig. 5.8b, ReorderWR(e_{11}, e_{12}) is applied on the execution graph G from Fig. 5.7b, which effectively swaps the order of e_{11} and e_{12}. We highlight the two events in blue. Note that the cycle we previously saw in Fig. 5.7b is resolved by reordering e_{11} and e_{12}. Since there is no other cycle in the execution graph, the execution graph obtained after the transformation is sequentially consistent. M2 is equipped with a MaxMin algorithm that linearlize a sequentially consistent execution into a well-formed trace. Fig. 5.8a shows such a trace output of the MaxMin algorithm. Note that in this trace, the two querying events, $w(z)$ and $r(z)$, occur consecutively. Hence, M2 is now able to report them as a data race.

5.2.3.2 Extending M2 to Predict TSO races

In this section, we formally introduce our extension to M2 for predicting TSO races. Algorithm 1 shows an extended algorithm for deciding whether a querying pair of events forms a predictable TSO race given a sequential trace σ. Intuitively, the algorithm is augmented with two procedures that remove events and orders in order to let the resulting closure to be sequentially consistent.

	t_1	t_2	t_3
1		$w(y)$	
2		acq(l)	
3		$w(x)$	
4			$r(x)$
5			$r(x)$
6		rel(l)	
7	acq(l)		
8	$w(x)$		
9	$r(y)$		
10			$w(y)$
11	**w(z)**		
12			**r(z)**

(a) A Trace output by the MaxMin algorithm of M2 after Transformation



(b) A Sequentially Consistent Execution

Figure 5.8: Transformed SC Execution

We highlight the two added steps in blue and the rest of the algorithm stays the same as the original RaceDecision from [Pav20]. The added steps are two new procedures that effectively transforms the witness execution. Algorithm 2 shows the procedure of ReadElimination and Algorithm 3 shows the procedure of RemoveNonPPO. We explain each of them in details below.

Algorithm 1 TSO Extension to RaceDecision [Pav20]

```
Input: A trace $\sigma$ and two events $e_1, e_2 \in \sigma$.Evts with $e_1 \bowtie e_2$.
Output: True if $(e_1, e_2)$ is detected as a predictable TSO race of $\sigma$.
 1: Let $X \leftarrow \mathsf{Rcone}_\sigma(e_1, \mathsf{p}(e_2)) \cup \mathsf{Rcone}_\sigma(e_2, \mathsf{p}(e_1))$
 2: if $\{e_1, e_2\} \cap X \neq \varnothing$ or $X$ is not feasible then return False
 3: $X \leftarrow \mathsf{ReadElimination}(X)$   ▷ Eliminate read events that reads from writes on the same thread
 4: Let $P \leftarrow R_\sigma(X)$
 5: $P \leftarrow \mathsf{RemoveNonPPO}(\sigma, P)$   ▷ Remove po orders that are not preserved under TSO
 6: Let $Q \leftarrow \mathsf{Closure}(\sigma, P, X)$
 7: if $Q = \perp$ then return False
 8: Non-deterministically choose $i \in [2]$
 9: while $\exists \bar{e_1}, \bar{e_2} \in X \setminus \mathsf{p}(e_i)$.Evts such that $\bar{e_1} \bowtie \bar{e_2}$ and $\bar{e_1} \|_Q \bar{e_2}$ and $\bar{e_1} <_\sigma \bar{e_2}$ do
10:     $Q \leftarrow \mathsf{InsertAndClose}(Q, \bar{e_1} \rightarrow \bar{e_2})$
11:     if $Q = \perp$ then return False
12: end
13: return True
```

Algorithm 2 ReadElimination

```
Input: A set of events $X$ of a well-formed trace
Output: An event set such that all read events that can be removed by RemoveWR are removed
 1: Let $E \leftarrow \varnothing$
 2: for $r \in X$.Rds do   ▷ Scan through each thread in the order of po
 3:     if $\exists w \in \mathsf{p}(r)$.Wrts s.t. $r.loc = w.loc$ and $r.val = w.val$ then
 4:         if $w.eid = r.eid - 1$ then
 5:             $E \leftarrow E \cup \{r\}$
 6:         else if $\forall e \in \mathsf{p}(r)$.Evts s.t. $w.eid < e.eid < r.eid$, either $e \in X$.Wrts or $e \in E$ then
 7:             $E \leftarrow E \cup \{r\}$
 8: Let $X' \leftarrow X \setminus E$
 9: return $X'$
```

Read Elimination From Section 5.2.1, we have seen a sound transformation, RemoveWR, which removes read events that read from write events on the same thread. Note that RemoveWR only corresponds to a *single* step of transformation. A desired sequentially consistent execution can be obtained from applying a sequence of this transformation, which may consists of multiple steps of RemoveWR and possibly interleaved with ReorderWR transformations. Therefore, Algorithm 2 recognizes *all* read events that can be removed by RemoveWR at some point in the sequence. Since the goal is to explain the behavior of the TSO-consistent witness execution in sequential consistency if such witness exists , ReadElimination removes all such read events to maximize the chance of sequential consistency. Let r be a read event in the input set such that there exists a write event w on the same thread as r where $w \xrightarrow{\text{po}} r$, $r.loc = w.loc$ and $r.val = w.val$. There are two cases when r is removable.

1. r immediately follows w on their thread and there is no other events between w and r. Note that $\langle w, r\rangle$ satisfies the conditions required by RemoveWR in this case. Therefore, r can be removed by RemoveWR. To identify this case, we check the eid of r and w since they are generated in the same order as po.

2. $w \xrightarrow{\text{po}} e \xrightarrow{\text{po}} r$ for some event e. Then r can still be removed by RemoveWR at some point if each of the events between w and r is either:

   (a) A write event to a different memory location. Since the input trace is well-formed and we assume each write event has a distinct value, we only need to check whether the event is a write event for this case. In this case, the write event may be reordered with r by ReorderWR, which may brings r to be immediately after w and makes it removeable by RemoveWR.

   (b) Or a read event that is also removable. Since the algorithm scan through the events following po, it's sufficient to scan the set once and there is no need for more iteration. In this case, removing the event can bring r to be immediately after w and makes it removeable by RemoveWR.

We demonstrate the above idea of in Fig. 5.9. In this figure, a thread with two write events and two write events is transformed into a thread with two write events in three steps. First, $r(y)$

$$\begin{tikzcd}
w(x) = 1 \arrow[d, dash] & & w(x) = 1 \arrow[d, dash] & & w(x) = 1 \arrow[d, dash] & & w(x) = 1 \arrow[d, dash] \\
w(y) = 1 \arrow[d, dash] \arrow[rr, "\mathsf{RemoveWR}(w(y), r(y))"] & & w(y) = 1 \arrow[d, dash] \arrow[rr, "\mathsf{ReorderWR}(w(y), r(x))"] & & r(x) = 1 \arrow[d, dash] \arrow[rr, "\mathsf{RemoveWR}(w(x), r(x))"] & & w(y) = 1 \arrow[d] \\
r(y) = 1 \arrow[d, dash] & & r(x) = 1 \arrow[d] & & w(y) = 1 \arrow[d] & & {} \\
r(x) = 1 \arrow[d] & & {} & & {} & & \\
{} & & & & & &
\end{tikzcd}$$

Figure 5.9: A transformation sequence that removes two read events

is removed since it immediately follows a write event $w(y)$ with the same value. Then $w(y)$ and $r(x)$ is reordered since they are accessing different locations and $r(x)$ immediately follows $w(y)$. Lastly, $r(x)$ is removed since now it imediately follows $w(x)$, and no further transformation is available after this step. Note that $r(x)$ only becomes removable after being reordered with $w(y)$, which is only possible after $r(y)$ is removed. Therefore, to identify such removable reads, we check the events between the write event and the read event. If all of the events in-between can be either reordered or removed at some point, then eventually the read event can move to be immediately following the write event and hence is eligible to be removed.

Algorithm 3 RemoveNonPPO

Input: A well-formed trace σ, and a partial order P over the events of σ such that $\sigma.\mathsf{po} \subseteq P$
Output: A partial order such that all $\langle e_1, e_2 \rangle$ that can be reordered by ReorderWR are removed

```
1: Let $E \leftarrow \varnothing$
2: **for** $\langle e_1, e_2 \rangle \in \sigma.\mathsf{po}$ **do**
3:     **if** $\langle e_1, e_2 \rangle \notin \sigma.(\mathsf{ppo} \cup \texttt{po-loc} \cup (\texttt{po}; [\texttt{L}]) \cup ([\texttt{L}]; \texttt{po}))$ **then**
4:         $E \leftarrow E \cup \{\langle e_1, e_2 \rangle\}$
5: Let $P' \leftarrow P \setminus E$
6: **return** $P'$
```

Write-to-Read Order Elimination Another sound transformation under TSO is ReorderWR. Just like RemoveWR, ReorderWR corresponds to a single step whereas a transformed execution graph may be the result of a sequence of multiple steps of transformations. In addition, M2 determines data race by inferring orders in an execution graph. Therefore, during the RaceDecision algorithm, we need to *remove* some orders and prevent M2 from inferring the orders that are not preserved under TSO. Algorithm 3 is a procedure of removing program orders that are not preserved under TSO, while keeping other orders in the closure. From Chapter 4, we have introduced the TSO model augmented with lock events. The augmented TSO model preserves all `ppo`, `po-loc`, and the program order with lock events. Hence, any po order that is not included in these three types are removed by the procedure of RemoveNonPPO

5.2.3.3 Soundness Proof

In this section, we prove the soundness of our extended algorithm, following the recepe from Chapter 4.

First, we show that ReadElimination is a sound transformation for TSO.

▶ Lemma 11
Let G be a plain execution graph over the set X before ReadElimination at line 3 of Algorithm 1, and G' be a plain execution graph over X after ReadElimination returns at line 3. Then there is a sequence of RemoveWR and ReorderWR transformations that transforms G to G', i.e., $G \rightsquigarrow G'$.

Proof. Let E be the set of read events removed by ReadElimination. That is, $G.\mathsf{Evts} = G'.\mathsf{Evts} \cup E$. According to Algorithm 2, for each $r \in E$, there exists a write event $w \in G.\mathsf{Wrts}$ on the same thread of r such that $r.val = w.val$ and $r.loc = w.loc$. There are two cases to distinguish.

- $\langle w, r\rangle \in G.\mathsf{po}|_{imm}$. Then $G' = G.\mathsf{RemoveWR}(w, r)$. Therefore, $G \rightsquigarrow G'$.
- $\forall e \in G.\mathsf{Evts}, w \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} r \Rightarrow (e \in G.\mathsf{Wrts} \wedge e.loc \neq r.loc) \vee (e \in E)$.
  - Base case. $\langle w, e\rangle, \langle e, r\rangle \in G.\mathsf{po}|_{imm}$. If $e \in E$, then $G' = G.\mathsf{RemoveWR}(e', e).\mathsf{RemoveWR}(w, r)$ for some e'. If $e \in G.\mathsf{Wrts}$ and $e.loc \neq r.loc$, then $G' = G.\mathsf{ReorderWR}(e, r).\mathsf{RemoveWR}(w, r)$. Therefore, $G \rightsquigarrow G'$.

· Inductive Step. $w \xrightarrow{\text{po}} e' \xrightarrow{\text{po}} e \xrightarrow{\text{po}|_{\text{imm}}} r$. IH: if $w \xrightarrow{\text{po}} e' \xrightarrow{\text{po}} r$ where $(e' \in G.\text{Wrts} \wedge e'.loc \neq r.loc) \vee (e' \in E)$, then $G'' \rightsquigarrow G'$ for some G''. If $e \in E$, then $G'' = G.\text{RemoveWR}(e'', e)$ for some e''. If $e \in G.\text{Wrts}$ and $e.loc \neq r.loc$, then $G'' = G.\text{ReorderWR}(e, r)$. Therefore, $G \rightsquigarrow G''$. By IH and the transitivity of $\rightsquigarrow$, $G \rightsquigarrow G'$.

□

We now prove Algorithm 1 is sound. Since we have already proved the soundness of the original version of M2 in Chapter 4, a large portion of the notations an existing proof are reused here.

▶ THEOREM 20 (SOUNDNESS OF EXTENDED M2)
If Algorithm 1 returns True, then $\langle e_1, e_2 \rangle$ is a sound data race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \text{RCone}_\sigma(e_1, e_2)$$

Then by the same way shown in the proof of THEOREM 4, let G_ρ be the plain execution graph constructed from S_σ and G_ρ is well-formed and executable for the same reasons as in the proof of THEOREM 4.

In addition to G_ρ, we construct a second plain execution graph G. Let S be a set defined as the following.

$$S = \text{ReadElimination}(S_\sigma)$$

and G is a plain execution graph constructed from S. Then by LEMMA 11, we can infer that that $G_\rho \rightsquigarrow G$.

▶ *Inserting Memory Orders.* We insert memory orders in G and show that there exists an insertion scheme such that G is TSO-consistent. Define the order

$$\texttt{preserve} \triangleq \texttt{ppo} \cup \texttt{po-loc} \cup (\texttt{po}; [\mathtt{L}]) \cup ([\mathtt{L}]; \texttt{po})$$

Then we use the same memory insertion scheme as in the proof of Theorem 4.

Following a similar reasoning process as in the proof of Theorem 4, we can see that

$$(\texttt{preserve} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{sync})^{+}$$

is irreflexive. Therefore, both $\texttt{ppo} \cup [\mathtt{L}]; \texttt{po} \cup \texttt{po}; [\mathtt{L}] \cup \texttt{com}$ and $\texttt{po-loc} \cup \texttt{com}$ are acyclic and G is TSO-consistent.

By Proposition 6, from $G_\rho \rightsquigarrow G$, we can conclude that G_ρ is TSO-consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, no further step is needed.

□

CHAPTER 6

Conclusion

We have shown that it is possible to formally prove the correctness of software tools under axiomatic memory models by reasoning about the execution graphs of programs. We demonstrated that Java can use a compilation scheme that is similar to C/C++11. On the other hand, one should not simply compile Java's Access Modes the same way as C/C++11 compiles atomic memory orders since the formal memory models supports different compiler optimizations. Moreover, we proposed a new modular soundness definition for predictive analyses under weak memory models. The new soundness definition subsumes all existing definitions and comes with a simple recipe for constructing a proof. Lastly, we extend an existing predictive analyses, which assumes sequential consistency, with sound transformations to predict new data races under a weak memory model.

APPENDIX A

the Full JAM21 Model

```
let opq = O | RA | V

let rel = W & (RA | V)

let acq = R & (RA | V)

let f_rel = REL | V

let f_acq = ACQ | V

let vol = V

let fence = F


(* volatile accesses extend push order *)

let svo = po;[fence & f_rel];po;[W] | [R];po;[fence & f_acq];po

let spush = po;[fence & vol];po


(* release acquire ordering *)

let ra = po;[rel] | [acq];po


(* intra thread volatile ordering *)

let volint =  [vol];po;[vol] (* OLD: po;[vol & R] | [vol & W];po *)


(* intrathread ordering contraints *)

let into = svo | spush | ra | volint


(* cross thread push ordering extended with volatile memory accesses *)

let push = spush | volint

with pushto from linearisations(domain(push), ((W * FW) & loc & ~id) | rf | po)
```

```
(* extend ra visibility *)
let vvo = rf | svo | ra | push | pushto;push
let vo = vvo+ | po-loc

include "filters.cat" (* includes WW filter *)
let WWco(rel) = WW(rel) & loc & ~id
(* final writes are co-after everything *)
let cofw = WWco((W * FW))

(* jam coherence *)
let coww = WWco(vo)
let cowr = WWco(vo;invrf)
let corw = WWco(vo;po)
let corr = [opq] ; WWco(rf;po;invrf) ; [opq]
let coinit = loc & IW*(W\IW)

include "cross.cat"
let co0 = loc & (IW * (W \ IW)|(W \ FW) * FW)
with cormwtotal from generate_orders(RMW, co0)

let rec co-jom = coww | cowr | corw | corr | cormwtotal
| WWco((rf;[RMW])^-1;co-jom) | coinit | cofw

acyclic (po | rf) ; [opq] as no-thin-air
acyclic co-jom as coherence
```

APPENDIX B

the Full JAM21' Model

```
include "filters.cat"
include "cross.cat"
let WWco(rel) = WW(rel) & loc & ~id
let co0 = loc & (IW * (W \ IW)|(W \ FW) * FW)
with cormwtotal from generate_orders(RMW, co0)

let opq = O | RA | V
let rel = W & (RA | V)
let acq = R & (RA | V)
let f_rel = REL | V
let f_acq = ACQ | V
let vol = V
let fence = F

(* volatile accesses extend push order *)
let svo = po;[fence & f_rel];po;[W] | [R];po;[fence & f_acq];po
let spush = po;[fence & vol];po

(* release acquire ordering *)
let ra = po;[rel] | [acq];po

(* intra thread volatile ordering *)
let volint =  [vol];po;[vol] (* OLD: po;[vol & R] | [vol & W];po *)
```

```
(* intrathread ordering contraints *)
let into = svo | spush | ra | volint
let push = spush | volint

let rec co-jom = coww | cowr | corw | corr | cormwtotal
| WWco((rf;[RMW])^-1;co-jom) | coinit | cofw

and fr-jom = rf^-1 ; co-jom
and fr-jom-e = fr-jom & ext
and co-jom-e = co-jom & ext
and chapo = rfe | fr-jom-e | co-jom-e | (fr-jom-e ; rfe) | (co-jom-e ; rfe)

(* extend ra visibility *)
and vvo = rf | svo | ra | push | push ; chapo ; push
and vo = vvo+ | po-loc

and cofw = WWco((W * FW))
and coww = WWco(vo)
and cowr = WWco(vo;invrf)
and corw = WWco(vo;po)
and corr = [opq] ; WWco(rf;po;invrf) ; [opq]
and coinit = loc & IW*(W\IW)

acyclic (po | rf) ; [opq] as no-thin-air
acyclic co-jom as coherence
```

APPENDIX C

The Power Memory Model in Herd7

```
PPC
(* Model for Power *)
include "cos.cat" (* Used to compute the coherence order*)

(* Uniproc *)
acyclic po-loc | rf | fr | co as scperlocation

(* Atomic *)
empty rmw & (fre;coe) as atomic

(* Utilities *)
let dd = addr | data
let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe ; rfe)
let addrpo = addr;po

(*******)
(* ppo *)
(*******)

let sync = try fencerel(SYNC) with 0
let lwsync = try fencerel(LWSYNC) with 0
let eieio = try fencerel(EIEIO) with 0
let isync = try fencerel(ISYNC) with 0
show sync,lwsync,eieio
```

```
(* Dependencies *)
show data,addr
let ctrlisync = try ctrlcfence(ISYNC) with 0
show ctrlisync
show isync  ctrlisync as isync
show ctrl  ctrlisync as ctrl
show isync,ctrlisync

(* Initial value *)
let ci0 = ctrlisync | detour
let ii0 = dd | rfi | rdw
let cc0 = dd | po-loc | ctrl | addrpo
let ic0 = 0

(* Fixpoint from i -> c in instructions and transitivity *)
let rec ci = ci0 | (ci;ii) | (cc;ci)
and ii = ii0 | ci | (ic;ci) | (ii;ii)
and cc = cc0 | ci | (ci;ic) | (cc;cc)
and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic) (* | ci inclus dans ii et cc *)

let ppo =
let ppoR = ii & (R * R)
and ppoW = ic & (R * W) in
ppoR | ppoW

(* fences *)

let lwsync = lwsync  (W * R)
let eieio = eieio & (W * W)

(* All arm barriers are strong *)
let strong = sync
let light = lwsync|eieio

let fence = strong|light

(* happens before *)
let hb = ppo | fence | rfe
acyclic hb as thinair

(* prop *)
let hbstar = hb*
let propbase = (fence|(rfe;fence));hbstar

let chapo = rfe|fre|coe|(fre;rfe)|(coe;rfe)

let prop = propbase & (W * W) | (chapo? ; propbase*; strong; hbstar)

acyclic co|prop as propagation
irreflexive fre;prop;hbstar as observation

let xx = po & (X * X)
acyclic co | xx as scXX
```

APPENDIX D

Proof of Observational Equivalence

▶ Proposition 7
Observational equivalence is a transitive relation. $G_1 \doteqdot G_2 \wedge G_2 \doteqdot G_3 \Rightarrow G_1 \doteqdot G_3$.

Proof. The transitivity follows directly from the transitivity of set equivalence in the definition of observational equivalence. □

▶ Lemma 12
($\Rightarrow$) Given a program P, for any $G \in [\![P]\!]_{\text{JAM21}}$, there exists a $G' \in [\![P]\!]_{\text{JAM21'}}$ such that $G' \doteqdot G$.

Proof. It's obvious that there exists a candidate execution G' that is observationally equivalent to G. We prove that $G' \in [\![P]\!]_{\text{JAM21'}}$. That is, let G' be a candidate execution of P and $G' \doteqdot G$. We show that G' is JAM21′-consistent.

Since the only difference between JAM21 and JAM21′ is at the effect of full fences, we focus on this part in our proof. In particular, we first show that, if $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$, then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$. We distinguish five cases by unfolding the `chapo` order and show that $i_2 \xrightarrow{\texttt{vo}} i_3$ leads to a contradiction in each case:

1. $i_3 \xrightarrow{\texttt{rf}} i_2$: then we have $i_1 \xrightarrow{\texttt{push}} i_3 \xrightarrow{\texttt{rf}} i_2 \xrightarrow{\texttt{push}} i_4$, which is equivalent to $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} i_2 \xrightarrow{\texttt{vvo}} i_4$, which means $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ (as it'd create a `vo` cycle in the latter case).

2. $i_3 \xrightarrow{\texttt{co}} i_2$: then it cannot be $i_2 \xrightarrow{\texttt{vo}} i_3$ on the right side as it immediately gives us a coherence cycle by coww. So it must be $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

3. $i_3 \xrightarrow{\texttt{fr}} i_2$: then there exists a write event w such that $w \xrightarrow{\texttt{rf}} i_3$ and $w \xrightarrow{\texttt{co}} i_2$. If we have $i_2 \xrightarrow{\texttt{vo}} i_3$ then we would have $i_2 \xrightarrow{\texttt{co}} w$ by cowr, which gives us a coherence cycle. Therefore it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

4. $i_3 \xrightarrow{\texttt{co}} w \xrightarrow{\texttt{rf}} i_2$ for some write event w: if we have $i_2 \xrightarrow{\texttt{vo}} i_3$ then, because $\texttt{rf} \subseteq \texttt{vo}$, we have $w \xrightarrow{\texttt{vo}} i_3$. By coww we get $w \xrightarrow{\texttt{co}} i_3$, contradicting with the earlier assumption that $i_3 \xrightarrow{\texttt{co}} w$. So it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

5. $i_3 \xrightarrow{\texttt{fr}} w_1 \xrightarrow{\texttt{rf}} i_2$ for some write event w_1: then it means there is w_2 such that $w_2 \xrightarrow{\texttt{rf}} i_3$ and $w_2 \xrightarrow{\texttt{co}} w_1$. If we have $i_2 \xrightarrow{\texttt{vo}} i_3$, since $\texttt{rf} \subseteq \texttt{vo}$, we have $w_1 \xrightarrow{\texttt{vo}} i_3$. By cowr, we have $w_1 \xrightarrow{\texttt{co}} w_2$, contradicting with the assumption of $w_2 \xrightarrow{\texttt{co}} w_1$ earlier. So it must be $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

Thus if we have $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$, then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

We now show that G' is JAM21$'$-consistent. Note that the NO-THIN-AIR requirement is obviously satisfied since G.Evts = G'.Evts, G.po $= G'$.po and $G.\texttt{rf} = G'.\texttt{rf}$, which follow from $G \mathrel{\overline{\overline{\mathrm{rf}}}} G'$.

Previously, we have shown that $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$ implies $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ in G. Since G is JAM21-consistent, then $\texttt{co-jom}$ is acyclic with either $i_1 \xrightarrow{\texttt{vo}} i_4$ or $i_2 \xrightarrow{\texttt{vo}} i_3$. Now we have two cases:

1. **No communication edge**: then we do not have any extra $\texttt{vo}$ edge we can use to infer in G' either. Since $\texttt{co-jom}$ is acyclic in G, and $G.\texttt{vo} = G'.\texttt{vo}$, $\texttt{co-jom}$ is acyclic in G', too.

2. **With communication** $\texttt{chapo}$: suppose $i_3 \xrightarrow{\texttt{chapo}} i_2$ (the other direction has a symmetrical proof), then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ in G. Since G is JAM21-consistent, we know that $i_1 \xrightarrow{\texttt{vo}} i_4$ cannot lead to any $\texttt{co-jom}$ cycle. In G', we can infer that $i_1 \xrightarrow{\texttt{vo}} i_4$. Since other portions of G' satisfies the above conditions, we can infer that $i_1 \xrightarrow{\texttt{vo}} i_4$ cannot lead to any $\texttt{co-jom}$ cycle in G' either.

Since neither case leads to a co-jom cycle in G', we can conclude that G' is JAM21$'$-consistent and hence $G' \in [\![P]\!]_{\text{JAM21}'}$

□

▶ LEMMA 13
Let fullfence $=$ push; chapo; push. In JAM21, for any co-jom cycle derived due to the fullfence orders, there is a vo*; fullfence; vo*; chapo cycle.

Proof. Let i_1, i_2, i_3 and i_4 be four events in an execution G such that $i_1 \xrightarrow{\texttt{push}} i_3$ and $i_2 \xrightarrow{\texttt{push}} i_4$. We can derive that $(i_1 \xrightarrow{\texttt{fullfence}} i_4) \vee (i_2 \xrightarrow{\texttt{fullfence}} i_3)$. Suppose that G is not consistent due to this condition under JAM21, i.e., following either side of the disjunction we can derive a co-jom cycle. We analyze one side of the disjunction since the other side of the disjunction symmetrically follow the same reasonings. We analyze each possible case to derive a coherence cycle. Since fullfence $\subseteq$ vo, we only need to analyze the cases where vo appears:

- coww. If we derived the coherence cycle from coww rule, then it means there exists w_1 and w_2 such that $w_1 \xrightarrow{\texttt{vo}} w_2$ and $w_2 \xrightarrow{\texttt{co}} w_1$. It implies the following structure: $w_1 \xrightarrow{\texttt{vo}}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}^* w_2 \xrightarrow{\texttt{co}} w_1$, revealing a cycle of vo*; fullfence; vo*; chapo.

- cowr. If we derived the coherence cycle from cowr rule, then it means there exists r_1, w_1, and w_2 such that $w_1 \xrightarrow{\texttt{rf}} r_1$, $w_2 \xrightarrow{\texttt{vo}} r_1$, and $w_1 \xrightarrow{\texttt{co}} w_2$. The fact that $i_1 \xrightarrow{\texttt{vo}} i_4$ "enables" us to derive this contradiction implies the following structure: $w_2 \xrightarrow{\texttt{vo}}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}^* r_1$. Because $w_1 \xrightarrow{\texttt{co}} w_2$ and $w_1 \xrightarrow{\texttt{rf}} r_1$, we have $r_1 \xrightarrow{\texttt{fr}} w_2$. We now have a cycle $w_2 \xrightarrow{\texttt{vo}}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}^* r_1 \xrightarrow{\texttt{fr}} w_2$, which is a cycle of vo*; fullfence; vo*; chapo.

- corw. If we derived the coherence cycle from corw rule, then it means there exists r_1, w_1, and w_2, such that $w_1 \xrightarrow{\texttt{rf}} r_1$, $r_1 \xrightarrow{\texttt{vo}} w_2$, and $w_2 \xrightarrow{\texttt{co}} w_1$. The fact that $i_1 \xrightarrow{\texttt{vo}} i_4$ "enables" us to derive this contradiction implies the following structure: $r_1 \xrightarrow{\texttt{vo}}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}^* w_2 \xrightarrow{\texttt{co}} w_1 \xrightarrow{\texttt{rf}} r_1$, which is a cycle of vo*; fullfence; vo*; chapo.

□

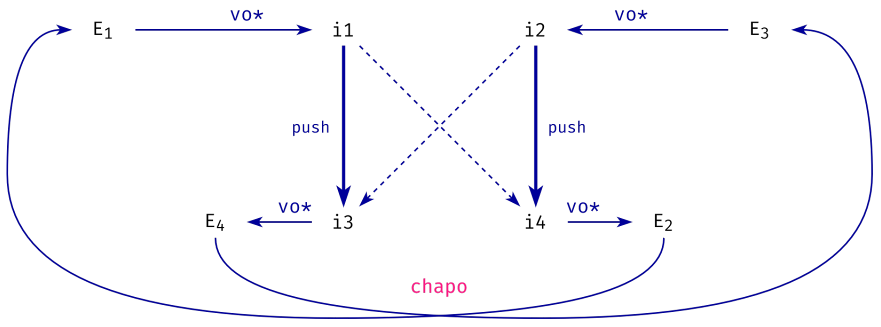


Figure D.1: Cycle Structure caused by Full Fences with Communications

▶ LEMMA 14
(⇐) Given a program P, for any JAM21′-consistent execution $G' \in [\![P]\!]_{\text{JAM21}'}$, there exists an execution history $G \in [\![P]\!]_{\text{JAM21}}$ such that $G \doteq G'$.

Proof. Given a program P, it's obvious that there exists an G that is observationally equivalent to G'. We prove that $G \in [\![P]\!]_{\text{JAM21}}$. That is, let G be a candidate execution of P and G is observationally equivalent to G'. We show that G is JAM21-consistent. To help the reader better understand this, consider Fig. D.1. Suppose G is an execution history that is forbidden by the rules of JAM21, we show that its corresponding G' is also forbidden by JAM21′. Since the only difference between JAM21 and JAM21′ is at the effects of full fences, we only analyze that part. In other words, G is forbidden by JAM21 precisely due to the total order of full fences. Let i_1, i_2, i_3, and i_4 be events in G such that $i_1 \xrightarrow{\text{push}} i_3$ and $i_2 \xrightarrow{\text{push}} i_4$. By Lemma 13, we can generalize the structure and infer that there are E_1, E_2, E_3, and E_4 such that $E_1 \xrightarrow{\text{vo}}{}^* i_1$, $E_3 \xrightarrow{\text{vo}}{}^* i_2$, $i_4 \xrightarrow{\text{vo}}{}^* E_2$, and $i_3 \xrightarrow{\text{vo}}{}^* E_4$. In addition, we also have $E_2 \xrightarrow{\text{chapo}} E_1$ and $E_4 \xrightarrow{\text{chapo}} E_3$. Because G is forbidden under JAM21, it means we have two cycles, $i_1 \xrightarrow{\text{vo}} i_4 \xrightarrow{\text{vo}}{}^* E_2 \xrightarrow{\text{chapo}} E_1 \xrightarrow{\text{vo}}{}^* i_1$ and $i_2 \xrightarrow{\text{vo}} i_3 \xrightarrow{\text{vo}}{}^* E_4 \xrightarrow{\text{chapo}} E_3 \xrightarrow{\text{vo}}{}^* i_2$, so that no matter which side of the disjunction we choose, we always end up with a contradiction. In G', on the other hand, we do not have $i_1 \xrightarrow{\text{vo}} i_4$ or $i_2 \xrightarrow{\text{vo}} i_3$. However, despite the absence of the two edges, we now have a larger cycle: $i_1 \xrightarrow{\text{push}} i_3 \xrightarrow{\text{vo}}{}^* E_4 \xrightarrow{\text{chapo}} E_3 \xrightarrow{\text{vo}}{}^* i_2 \xrightarrow{\text{push}} i_4 \xrightarrow{\text{vo}}{}^* E_2 \xrightarrow{\text{chapo}} E_1$, which forms a vo cycle by (vo-5') in JAM21′. Therefore, execution history G' is forbidden under JAM21′ as well, which contradicts to our previous assumption. Thus, since $G' \in [\![P]\!]_{\text{JAM21}'}$ implies that G' is JAM21′-consistent, $G \in [\![P]\!]_{\text{JAM21}}$. ☐

Essentially, the difference between JAM21 and JAM21′ give no actual effect in forbidding executions. In JAM21, we look for two `vo` cycles to forbid an execution, whereas in JAM21′ we combine the two cycles into one to forbid the execution.

We can now show the observational equivalence between the two models.

▶ THEOREM 1
JAM21′ $\doteqdot$ JAM21.

Proof. Given a program P, let $[\![P]\!]_{\text{JAM21}}$ be the set of JAM21-consistent candidate executions of P and $[\![P]\!]_{\text{JAM21}'}$ be the set of JAM21′-consistent candidate executions of P. By Lemma 12, we know that for all $G \in [\![P]\!]_{\text{JAM21}}$, there exists an $G' \in [\![P]\!]_{\text{JAM21}'}$ such that G' and G are observationally equivalent. Similarly, by Lemma 14, we know that for all $G' \in [\![P]\!]_{\text{JAM21}'}$, there exists an $G \in [\![P]\!]_{\text{JAM21}}$ such that G and G' are observationally equivalent. Combining together, we can conclude that JAM21 and JAM21′ are observationally equivalent. ☐

▶ COROLLARY 4
JAM21′ satisfies the same important properties (Theorem 21, Theorem 22, Theorem 23, Theorem 24 and Corollary 5) in Section. H.

Proof. Since the definitions in JAM21′ are the same as JAM21 except for the semantics of `fullFences`, JAM21′ automatically satisfies Theorem. 21, Theorem. 22, Theorem 23, and Theorem. 24. By Theorem. 1, JAM21 and JAM21′ allow the same set of execution up to observational equivalence. Therefore, JAM21′ also satisfies Corollary. 5. ☐

APPENDIX E

Proof of Compilation Correctness to Power

▶ LEMMA 1 (JAM21' TO POWER)
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (with the leading fence convention). For all $G_{tgt} \in [\![P_{tgt}]\!]_{\text{POWER}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21'}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Proof. It is obvious that there exists a candidate execution history G_{src} of P_{src} such that $G_{src} \rightsquigarrow G_{tgt}$. We show that $G_{src} \in [\![P]\!]_{\text{JAM21}'}$. That is, G_{src} is JAM21′-consistent. In order to be consistent under JAM21′, we need G_{src} to satisfy two requirements:

1. NO-THIN-AIR Requirement: $(\text{po} \mid \texttt{rf})$ is acyclic. The intra-thread `rfi` order cannot contradicting the po given that G_{tgt} is Power-consistent. Therefore, we need to show that $(\text{po} \mid \texttt{rfe})$ is acyclic. Note that since the only inter-thread order is `rfe`. If there is a cycle in $(\text{po} \mid \texttt{rfe})$, then the head of each thread is a r^{Opq} and the last event in each thread participating in this cycle is a w^{Opq}, where $r^{\text{Opq}} \xrightarrow{\text{po}} w^{\text{Opq}}$ in G_{src}. Using a compiler that follows the compilation scheme, this translates to $r \xrightarrow{\texttt{ctrl}} w$ in G_{tgt}. Further we can infer that $r \xrightarrow{\texttt{hb}} w$ in G_{tgt}. Power ensures that $(\texttt{hb} \mid \texttt{rf})$ is acyclic. Therefore, if there is a cycle of $(\text{po} \mid \texttt{rf})$ in G_{src}, G_{tgt} would not be consistent under Power's memory model, contradicting to our previous assumption. For readers who do not care about this guarantee, `getOpaque()` can be directly compiled to a `lwz` instruction.

2. COHERENCE Requirement: `co-jom` is acyclic. We now prove that `co-jom` is acyclic in G_{src}. In order to show this, we show that `co-jom` is a partial order of `co` in G_{tgt}. In other words,

`co` is a linear extension of `co-jom`. We prove this by assuming the opposite and deriving a contradiction.

Suppose that there exists $i_1 \xrightarrow{\texttt{co-jom}} i_2$ in G_{src} but $i_2 \xrightarrow{\texttt{co}} i_1$ in G_{tgt}. We analyze each of the possible cases where we can derive a `co-jom` order.

- coinit. This automatically gives us a contradiction since $G_{src}.\texttt{IW} = G_{tgt}.\texttt{IW}$.
- cofw. This automatically gives us a contradiction since $G_{src}.\texttt{FW} = G_{tgt}.\texttt{FW}$.
- corr. This implies that there exist r_1 and r_2 such that, $r_1 \xrightarrow{\texttt{po}} r_2$, $i_1 \xrightarrow{\texttt{rf}} r_1$, and $i_2 \xrightarrow{\texttt{rf}} r_2$. We also have $i_2 \xrightarrow{\texttt{co}} i_1$. Now the SC-per-location requirement of Power is violated, contradicting with our previous assumption that G_{tgt} is Power-consistent.
- coww. This implies that $i_1 \xrightarrow{\texttt{vo}} i_2$ but $i_2 \xrightarrow{\texttt{co}} i_1$.
  - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making G_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{svo}} i_2$ or $i_1 \xrightarrow{\texttt{ra}} i_2$ or $i_1 \xrightarrow{\texttt{push}} i_2$. In G_{tgt}, this means $i_1 \xrightarrow{\texttt{lwsync}} i_2$ or $i_1 \xrightarrow{\texttt{sync}} i_2$. Note that all three cases of them are included in program order with access to the same location. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making G_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{push}} e_1 \xrightarrow{\texttt{chapo}} e_2 \xrightarrow{\texttt{push}} i_2$ for some e_1 and e_2. Note that we have $e_1 \xrightarrow{\texttt{chapo}} e_2 \xrightarrow{\texttt{sync}} i_2$ and $i_2 \xrightarrow{\texttt{co}} i_1 \xrightarrow{\texttt{sync}} e_1$. They form a propagation (`prop`) cycle between i_2 and e_1, which makes G_{tgt} not Power-consistent, giving us a contradiction.
  - $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} i_2$. This corresponds to the inductive case where the visibility order is formed by two visibility orders through another event, i_3. Note that all the cases of `vo` orders produce propagation (`prop`) orders. Therefore, we get a violation of the Propagation requirement if $i_2 \xrightarrow{\texttt{co}} i_1$ in Power, contradicting to our previous assumption that G_{tgt} is Power-consistent.
- cowr. This implies that there exists r such that $i_2 \xrightarrow{\texttt{rf}} r$ and $i_1 \xrightarrow{\texttt{vo}} r$.

  - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-PER-LOCATION requirement of Power, making G_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{svo}} r$ or $i_1 \xrightarrow{\texttt{ra}} r$ or $i_1 \xrightarrow{\texttt{push}} r$. In G_{tgt}, this means $i_1 \xrightarrow{\texttt{lwsync}} i_2$ or $i_1 \xrightarrow{\texttt{sync}} i_2$. Note that all three cases of them are included in program order with access to the same location. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-PER-LOCATIONrequirement of Power, making G_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{push}} e_1 \xrightarrow{\texttt{chapo}} e_2 \xrightarrow{\texttt{push}} r$ for some e_1 and e_2. Because $i_2 \xrightarrow{\texttt{rf}} r$ and $i_2 \xrightarrow{\texttt{co}} i_1$, we have $r \xrightarrow{\texttt{fr}} i_1$. We now have $e_1 \xrightarrow{\texttt{chapo}} e_2 \xrightarrow{\texttt{sync}} r$ and $r \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{sync}} e_1$. Both of them form a propagation order between r and e_1 and they contradict with each other.
  - $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} r$ This corresponds to the inductive case where the visibility order is formed by two visibility orders through another event, i_3. Note that all the cases of `vvo` orders produce propagation orders. Therefore, we get $r \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{prop}} r$, which violates the OBSERVATION requirement in Power.

- corw. This implies that there exists r such that $r \xrightarrow{\texttt{po}} i_2$ and $i_1 \xrightarrow{\texttt{rf}} R$. $i_2 \xrightarrow{\texttt{co}} i_1$ would cause a violation of SC-PER-LOCATION requirement in Power.
- cormwexcl. This implies that i_1 is a rmw operation and there exists i_3 such that $i_3 \xrightarrow{\texttt{rf}} i_1$ and $i_3 \xrightarrow{\texttt{co}} i_2$. Having $i_2 \xrightarrow{\texttt{co}} i_1$ immediately violates the ATOMICITY requirement of Power.
- cormwtotal. $i_1 \xrightarrow{\texttt{co}} i_2$ because $G_{tgt}.\texttt{co} \subseteq G_{src}.\texttt{to}$. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ in this case would yield a cycle in `co`, violating the Propagation requirement in Power.

Thus, we have shown that `co-jom` is a partial order of `co` in G_{tgt} and the coherence requirement is automatically fulfilled because `co` is acyclic. Hence, $G_{src} \in [\![P]\!]_{\text{JAM21}'}$.

□

▶ **Theorem 2** (Compilation Correctness to Power (Leading Fence))
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (using the leading fence convention). For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{Power}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Proof. By Lemma 1, we know that there exists an $G'_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$, such that $G'_{src} \rightsquigarrow G_{tgt}$. Therefore, by definition of the $\rightsquigarrow$ relation,

- G_{tgt} is observationally equivalent to G'_{src}
- $G_{tgt}.\texttt{co} \subseteq G'_{src}.\texttt{to}$
- If $rmw, i_1 \in G'_{src}$.Evts and $rmw \xrightarrow{\texttt{po}} i_1$, then $rmw \xrightarrow{\texttt{ctrl}} i_1$ in G_{tgt}
- If $r^{\text{Opq}}, i_1 \in G'_{src}$.Evts and $r^{\text{Opq}} \xrightarrow{\texttt{po}} i_1$, then $r \xrightarrow{\texttt{ctrl}} i_1$ in G_{tgt}
- If $i_1, i_2 \in G'_{src}$.Evts and $i_1 \xrightarrow{\texttt{push}} i_2$, then $i_1 \xrightarrow{\texttt{sync}} i_2$ for $i_1, i_2 \in G_{tgt}$.Evts
- If $i_1, i_2 \in G'_{src}$.Evts and $i_1 \xrightarrow{\texttt{ra}} i_2$, then $i_1 \xrightarrow{\texttt{lwsync}} i_2$ for $i_1, i_2 \in G_{tgt}$.Evts

By Theorem 1, we know that for all G'_{src}, there exists an $G_{src} \in [\![P]\!]_{\text{JAM21}}$ such that G_{src} is observationally equivalent to G'_{src}. By Lemma 7, G_{src} is observationally equivalent to G_{tgt}. Furthermore,

- $G_{tgt}.\texttt{co} \subseteq G_{src}.\texttt{to}$ because $G_{src}.\texttt{to} = G'_{src}.\texttt{to}$.
- If $rmw, i_1 \in G_{src}$.Evts and $rmw \xrightarrow{\texttt{po}} i_1$, then $rmw \xrightarrow{\texttt{ctrl}} i_1$ in G_{tgt} because G_{src}.Evts $=$ G'_{src}.Evts and $G_{src}.\texttt{po} = G'_{src}.\texttt{po}$.
- If $r^{\text{Opq}}, i_1 \in G_{src}$.Evts and $r^{\text{Opq}} \xrightarrow{\texttt{po}} i_1$, then $r \xrightarrow{\texttt{ctrl}} i_1$ in G_{tgt} because G_{src}.Evts $= G'_{src}$.E and $G_{src}.\texttt{po} = G'_{src}.\texttt{po}$.
- If $i_1, i_2 \in G_{src}$.Evts and $i_1 \xrightarrow{\texttt{push}} i_2$, then $i_1 \xrightarrow{\texttt{sync}} i_2$ for $i_1, i_2 \in G_{tgt}$.Evts because $\forall i \in G_{src}$.Evts, $G_{src}.AccessMode(i) = G'_{src}.AccessMode(i)$ and $G_{src}.\texttt{po} = G'_{src}.\texttt{po}$, which means $G_{src}.\texttt{push} = G'_{src}.\texttt{push}$.

- If $i_1, i_2 \in G_{src}$.Evts and $i_1 \xrightarrow{\texttt{ra}} i_2$, then $i_1 \xrightarrow{\texttt{lwsync}} i_2$ for $i_1, i_2 \in G_{tgt}$.Evts because $\forall i \in G_{src}$.Evts, $G_{src}.AccessMode(i) = G'_{src}.AccessMode(i)$ and G_{src}.po $= G'_{src}$.po, which means $G_{src}.\texttt{ra} = G'_{src}.\texttt{ra}$.

Therefore, we have shown that for all $G_{tgt} \in [\![P]\!]_{\text{Power}}$, there exists an $G_{src} \in [\![P]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$. That is, the compilation scheme shown in Fig. 3.5 is correct.

□

▶ Corollary 1 (Compilation Correctness to Power (Trailing Fence))
Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 3.5 (using the trailing fence convention). For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{Power}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Proof. It is obvious that all the properties described in Definition 7 still hold with the trailing fence convention. Most importantly, the property that transforms `push` in the source level execution to `sync` in the target level execution is preserved as long as there is a `hwsync` instruction inserted between every Volatile accesses. The trailing fence convention, if used consistently, clearly satisfy this property. Then the rest of Definition 7 is unchanged since leading/trailing fence convention only concerns the compilation schemes for Volatile accesses. Therefore, the rest of the proof for the correctness of the trailing fence convention can be naturally derived similarly. □

APPENDIX F

The x86 TSO Model in Herd7

```
X86 TSO

include ”x86fences.cat”

include ”filters.cat”

include ”cos.cat”


(* Uniproc check *)

let com = rf | fr | co

acyclic po-loc | com


(* Atomic *)

empty rmw & (fre;coe)


(* GHB *)

#ppo

let po_ghb = WW(po) | RM(po)


#mfence

let mfence = try fencerel(MFENCE) with 0

let lfence = try fencerel(LFENCE) with 0

let sfence = try fencerel(SFENCE) with 0


show data,addr,ctrl


#implied barriers

let poWR = WR(po)
let i1 = MA(poWR)
let i2 = AM(poWR)
let implied = i1 | i2

let ghb = mfence | implied | po_ghb | rfe | fr | co
show implied
acyclic ghb as tso
```

APPENDIX G

Proof of Compilation Correctness to x86-TSO

▶ LEMMA 2
Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{TSO}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}'}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Proof. It is obvious that there exists an G_{src} such that $G_{src} \rightsquigarrow G_{tgt}$. We show that $G_{src} \in [\![P]\!]_{\text{JAM21}'}$. That is, we show that G_{src} is consistent under JAM21′ by showing that it fulfills the two requirements of JAM21′.

1. NO-THIN-AIR. $(\mathtt{po}|\mathtt{rf})$ is acyclic. The $\mathtt{rfi}$ order is included in the $\mathtt{po}$. Therefore, we need to show that $(\mathtt{po}|\mathtt{rfe})$ is acyclic. Note that since the only inter-thread order is $\mathtt{rfe}$. If there is a cycle in $(\mathtt{po}|\mathtt{rfe})$, then the head of each thread is a r^{Opq} and the last event in each thread participating in this cycle is a w^{Opq}, where $r^{\mathsf{Opq}} \xrightarrow{\mathtt{po}} w^{\mathsf{Opq}}$ in G_{src}. Using a compiler that follows the compilation scheme, this translates to $r \xrightarrow{\mathtt{po}} w$ in G_{tgt}. Further we can infer that $r \xrightarrow{\mathtt{ghb}} w$ in G_{tgt}. x86-TSO ensures that $\mathtt{ghb}$ is acyclic. Therefore, if there is a cycle of $(\mathtt{po}|\mathtt{rf})$ in G_{src}, G_{tgt} would not be consistent under x86-TSO, contradicting to our previous assumption.

2. COHERENCE. In order to show that G_{src} fulfills the coherence requirement, we need to show that $\mathtt{co}$ in G_{tgt} is a linear extension of $\mathtt{co\text{-}jom}$ in G_{src}. We prove this by analyzing each case for $\mathtt{co\text{-}jom}$. That is, if $i_1 \xrightarrow{\mathtt{co\text{-}jom}} i_2$ but $i_2 \xrightarrow{\mathtt{co}} i_1$, then G_{tgt} is inconsistent under x86-TSO.

- coinit. This follows naturally as $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$.

- cofw. This follows naturally as $G_{src}.\texttt{FW} = G_{tgt}.\texttt{FW}$.
- corr. This implies that there exists r_1 and r_2 such that $i_1 \xrightarrow{\texttt{rf}} r_1$, $i_2 \xrightarrow{\texttt{rf}} r_2$, and $r_1 \xrightarrow{\texttt{po}} r_2$. From $i_2 \xrightarrow{\texttt{co}} i_1$ we can infer that $r_2 \xrightarrow{\texttt{fr}} i_1$. Note that $r_1 \xrightarrow{\texttt{ghb}} r_2$ in this case since a po order from a read event is preserved in TSO. Now we have a ghb cycle $r_2 \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{rf}} r_1 \xrightarrow{\texttt{ghb}} r_2$, contradicting to out previous assumption that G_{tgt} is consistent under x86-TSO.
- coww. This implies that $i_1 \xrightarrow{\texttt{vo}} i_2$. We analyze each case of vo order.
  - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making G_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{ra}} i_2$ or $i_1 \xrightarrow{\texttt{push}} i_2$ or $i_1 \xrightarrow{\texttt{svo}} i_2$. Note that all three cases are included in $i_1 \xrightarrow{\texttt{po}} i_2$ to the same location. Thus, if $i_2 \xrightarrow{\texttt{co}} i_1$, then there would be a cycle of $(\texttt{po-loc}|\texttt{com})$ in G_{tgt}, contradicting to our previous assumption that G_{tgt} is consistent under TSO.
  - $i_1 \xrightarrow{\texttt{push}} e_1 \xrightarrow{\texttt{chapo}} e_2 \xrightarrow{\texttt{push}} i_2$ for some e_1 and e_2 in G_{src}. Since i_1 is a write, we can infer that $i_1 \xrightarrow{\texttt{po}} i_{\textsf{LOCK}} \xrightarrow{\texttt{po}} e_1$ in G_{tgt}, where $i_{\textsf{LOCK}}$ is a RMW event. According to x86-TSO, we can further infer that $i_1 \xrightarrow{\texttt{ghb}} i_{\textsf{LOCK}} \xrightarrow{\texttt{ghb}} e_1$ in G_{tgt}, which can be simplified to $i_1 \xrightarrow{\texttt{ghb}} e_1$. Similarly, we can infer that $e_2 \xrightarrow{\texttt{ghb}} i_2$ (if e_2 is a read then the po order is included in ghb; otherwise, there must be a RMW event between e_2 and i_2, which yields a ghb order too). Now, since the communication edges induced by chapo are also included in ghb in G_{tgt}, $i_2 \xrightarrow{\texttt{co}} i_1$ would directly produce a ghb cycle, contradicting with our previous assumptions.
  - $i_1 \xrightarrow{\texttt{vvo}} E \xrightarrow{\texttt{vvo}} i_2$ for some E in G_{src}. Note that vo orders in G_{src} only produce ghb orders in G_{tgt}. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would always result in a ghb cycle in G_{tgt}, contradicting to the previous assumption.
- cowr. This implies that $i_1 \xrightarrow{\texttt{vo}} r$ and $i_2 \xrightarrow{\texttt{rf}} r$ for some $r \in G_{src}$.Rds. With $i_2 \xrightarrow{\texttt{co}} i_1$ we can infer that $r \xrightarrow{\texttt{fr}} i_1$. As in previous cases, we observe that vo only produce ghb in G_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ would result in a ghb cycle in G_{tgt}.

- corw. This implies that there is r in G_{src} such that $r \xrightarrow{\texttt{po}} i_2$ and $i_1 \xrightarrow{\texttt{vo}} r$. Note that po order from a Read access is included in ghb in G_{tgt}. As in previous cases, we observe that vo only produce ghb in G_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ would result in a ghb cycle, contradicting to our previous assumption.
- cormwexcl. This implies that $w \xrightarrow{\texttt{rf}} r, r \xrightarrow{\texttt{rmw}} i_1$, and $w \xrightarrow{\texttt{co}} i_2$ for some w and r in G_{src}. We also have $r \xrightarrow{\texttt{fr}} i_2$. Now having $i_2 \xrightarrow{\texttt{co}} i_1$ would violate the Atomicity Requirement of TSO, contradicting to our previous assumptions.
- cormwtotal. Since there is no other restrictions on the total order among RMW operations except that it has to be compatible with the rf and intra-thread orders, it automatics becomes a subset of co in G_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ in this case would yield a cycle in co, producing a ghb cycle in G_{tgt}.

Thus we have shown that co is a linear extension of co-jom. As a consequence, co-jom is guaranteed to be acyclic is G_{tgt} is consistent under TSO.

Thus we can conclude that $G_{src} \in [\![P]\!]_{\text{JAM21}'}$. □

▶ Theorem 3
Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For each $G_{tgt} \in [\![P_{tgt}]\!]_{\text{TSO}}$ there exists a $G_{src} \in [\![P_{src}]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$.

Proof. By Lemma 2, we know that there exists an $G'_{src} \in [\![P]\!]_{\text{JAM21}'}$ such that $G'_{src} \rightsquigarrow G_{tgt}$. Therefore, by definition of the $\rightsquigarrow$ relation,

1. $G_{tgt}.\text{Evts} \backslash G_{tgt}.\text{F} = G'_{src}.\text{Evts} \backslash G'_{src}.\text{F}$, $G_{tgt}.\texttt{rf} = G'_{src}.\texttt{rf}$, and $G'_{src}.\texttt{po} \subseteq G_{tgt}.\texttt{po}$
2. $G_{tgt}.\texttt{co} \subseteq G'_{src}.\texttt{to}$
3. If $i_1, i_2 \in G'_{src}.\text{Evts}$ and $i_1 \xrightarrow{\texttt{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2$ for $i_1, i_2 \in G_{tgt}.\text{Evts}$ and $i_3 \in G_{tgt}.\text{F}$ where i_3 is an event stem from an `mfence` instruction.

By Theorem 1, we know that for all G'_{src}, there exists an $G_{src} \in [\![P]\!]_{\text{JAM21}}$ such that G_{src} is observationally equivalent to G'_{src}. Therefore, we have $G_{tgt}.\texttt{Evts} \backslash G_{tgt}.\texttt{F} = G_{src}.\texttt{Evts} \backslash G_{src}.\texttt{F}$, $G_{tgt}.\texttt{rf} = G_{src}.\texttt{rf}$, and $G_{src}.\texttt{po} \subseteq G_{tgt}.\texttt{po}$

Furthermore,

1. $G_{tgt}.\texttt{co} \subseteq G_{src}.\texttt{to}$ since $G_{src}.\texttt{to} = G'_{src}.\texttt{to}$.

2. If $i_1, i_2 \in G_{src}.\texttt{Evts}$ and $i_1 \xrightarrow{\texttt{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2$ for $i_1, i_2 \in G_{tgt}.\texttt{Evts}$ and $i_3 \in G_{tgt}.\texttt{F}$ where i_3 is an event stem from an `mfence` instruction, because $\forall i \in G_{src}.\texttt{Evts}, G_{src}.AccessMode(i) = G'_{src}.AccessMode(i)$ and $G_{src}.\texttt{po} = G'_{src}.\texttt{po}$, which means $G_{src}.\texttt{push} = G'_{src}.\texttt{push}$.

Therefore, we have shown that for all $G_{tgt} \in [\![P]\!]_{\text{TSO}}$, there exists an $G_{src} \in [\![P]\!]_{\text{JAM21}}$ such that $G_{src} \rightsquigarrow G_{tgt}$. That is, the compilation scheme to x86 is correct. □

APPENDIX H

Key Properties of the JAM21 Model

In this section, we show some key properties of JAM21. First we show that the prior theorems of JAM19 still hold for JAM21 in Section H.1. Then in Section H.2, we prove that when all accesses in program order are `push` ordered then the semantics of executions is sequentially consistent. As a corollary when all accesses are Volatile, which implies a `push` order, then the executions are sequentially consistent. We have defined and proved these theorems in Coq. The Coq source code is included in our supplementary materials.

H.1 Prior Theorems

The JAM21 model preserves the two main theoretical results of [BP19], namely the monotonicity of access modes and the causal-acquire reads. We recount each theorem here briefly beginning with the monotonicity of access modes using the same notation from [BP19].

We use the reflexive ordering of the access modes as Plain $\sqsubseteq$ Opaque $\sqsubseteq$ Release Acquire $\sqsubseteq$ Volatile and extend it to accesses $l_{m_1} \sqsubseteq l_{m_2}$, $l_{m_1} := n_1 \sqsubseteq l_{m_2} := n_2$, $\mathsf{RW}(l, n_1) \sqsubseteq \mathsf{RW}(l, n_2)$ whenever $m_1 \sqsubseteq m_2$. We treat read-modify-write (RMW) events as always having the same order. We extend the order to histories by matching identifiers and ordering the accesses.

$$H_1 \sqsubseteq H_2 \triangleq \forall\, i\, a_1\, a_2, H_1(\mathsf{is}(i, a_1)) \wedge H_2(\mathsf{is}(i, a_2)) \Rightarrow a_1 \sqsubseteq a_2$$

We adopt the same notion of "well-formedness" from [BP19] for a given history H, i.e., *trace* *coherence*[1].

▶ Definition 25 (Trace Coherence)
An execution history H is trace coherent if:

- Each memory location is initialized by an initial write. For each event $i \in H.\mathtt{E} \setminus \{H.\mathtt{IW} \cup H.\mathtt{F}\}$, there exists an initial write event $iw \in H.\mathtt{IW}$ such that $H.loc(i) = H.loc(iw)$ and $iw \xrightarrow{\mathtt{to}} i$.
- Reads-from edges are well-formed. For all $r \in H.\mathtt{R}$, there exists a unique write w such that $H.loc(w) = H.loc(r)$, $H.val(w) = H.val(r)$, and $w \xrightarrow{\mathtt{rf}} r$.
- There exists a total trace order `to` for all $e \in H.\mathtt{E}$ such that `to` is compatible with po, `rf`, `ra`, `svo`, and `push`.

When the po, `rf`, and to relations of two histories H_1 and H_2 have the following relationships: $H_2.\mathtt{po} \subseteq H_1.\mathtt{po}$, $H_2.\mathtt{to} \subseteq H_1.\mathtt{to}$, $H_2.\mathtt{rf} \subseteq H_1.\mathtt{rf}$, then we say they *match*.

▶ Theorem 21 (Monotonicity)
[`coq/Monotonicity.v, monotonicity`]
For two histories H_1 and H_2, suppose that both match, both are trace coherent, and $H_2 \sqsubseteq H_1$. Further suppose that acyclic($\xrightarrow{\mathtt{co}}_{H_1}$) and that there are no specified visibility orders or `push` orders in H_2, then acyclic($\xrightarrow{\mathtt{co}}_{H_2}$)

A version of DRF-SC theorem was proved in [BP19]. However, the theorem was different from the standard DRF-SC theorem.

- It did not use the conventional definition of data race with the "happens-before" order. Instead. [BP19] defined a sync order that captures the synchronizations between events and defined the notion of "data-race-free" using sync.
- It used a stronger assumption than the standard DRF-SC theorem. In particular, given a program P, the standard DRF-SC theorem assumes only the SC-consistent executions of P are data race free. On the other hand, the DRF-SC theorem proved in [BP19] assumes all executions of

[1]We have omitted some of the details of trace coherence that are related to the internals of the modeling language as they are irrelevant here

P are data race free. A similar theorem was also proved in [WPP20], called a "model-agnostic" definition of DRF-SC.

Here, we first prove the standard DRF-SC theorem (DRF-SC) with a weaker assumption than [BP19], and then prove a weaker DRF-SC theorem (Execution-DRF), both using "happens-before" (hb).

We require the following standard definitions including the traditional notion of sequential consistency (SC-consistency) [Lam79]:

$$i_1 \xrightarrow{\texttt{fr}} i_2 \triangleq \exists\, i_3, i_3 \xrightarrow{\text{rf}} i_1 \wedge i_3 \xrightarrow{\texttt{co}} i_2$$
$$i_1 \xrightarrow{\text{com}} i_2 \triangleq i_1 \xrightarrow{\texttt{co}} i_2 \vee i_1 \xrightarrow{\text{rf}} i_2 \vee i_1 \xrightarrow{\text{fr}} i_2$$
$$i_1 \xrightarrow{\text{sc}} i_2 \triangleq i_1 \xrightarrow{\text{po}} i_2 \vee i_1 \xrightarrow{\text{com}} i_2$$

An execution H is SC-consistent if acyclic$(\xrightarrow{\text{sc}}_H)$.

We also require the notion of *happens-before* (hb) defined using the *synchronizes-with* (sw) order:

$$\begin{aligned} i_1 \xrightarrow{\texttt{sw}} i_2 \triangleq\ & (i_1 \xrightarrow{\texttt{rf}} i_2 \wedge AccessMode(i_1) = \texttt{Release} \wedge AccessMode(i_2) = \texttt{Acquire}) \\ & \vee (\exists i_3 i_4 \in \texttt{F}, AccessMode(i_3) = \texttt{Release} \wedge AccessMode(i_4) = \texttt{Acquire} \\ & \wedge i_3 \xrightarrow{\texttt{po}} i_1 \xrightarrow{\texttt{rf}} i_2 \xrightarrow{\texttt{po}} i_4) \end{aligned}$$
$$i_1 \xrightarrow{\texttt{hb}} i_2 \triangleq i_1 \xrightarrow{\texttt{po}} i_2 \vee i_1 \xrightarrow{\texttt{sw}} i_2 \vee \exists i_3, i_1 \xrightarrow{\texttt{hb}} i_3 \xrightarrow{\texttt{hb}} i_2$$

▶ Definition 26
Two memory accesses i_1 and i_2 are **conflicting** in an execution H if:

- $i_1, i_2 \in H.\mathsf{E}$
- $H.loc(i_1) = H.loc(i_2)$
- At least one of i_1 and i_2 is a write

▶ Definition 27
Two memory accesses i_1 and i_2 form a **data race** if:

- i_1 and i_2 are conflicting
- $\neg(i_1 \xrightarrow{\text{hb}} i_2 \vee i_2 \xrightarrow{\text{hb}} i_1)$

We say they form a **volatile-race** if both i_1 and i_2 are Volatile mode accesses.

Finally, our DRF-SC theorem is stated as the following:

▶ Theorem 22 (DRF-SC)
Given a program P, if all its SC-consistent executions are data-race-free or only have volatile-races, then the set of all Jam21-consistent executions of P coincide with the set of SC-consistent executions.

Please see Appendix I for the proof.

We also provide a weaker version of the DRF-SC theorem:

▶ Theorem 23 (Execution-drf)
Any Jam21-consistent execution that is data race free or only has volatile-races is SC-consistent.

Please see Appendix J for the proof.

Finally, we demonstrate the revised semantics preserves causality with acquire reads.

▶ Theorem 24 (Causal Acquire-Reads)
[coq/ReleaseAcquire.v, acq_causality]
If H is trace coherent and all reads in H are acquire-reads, then acyclic($\xrightarrow{\text{po}|\text{rf}}$).

H.2 Volatile implies SC

Here we demonstrate that when all accesses are volatile, the program will have SC semantics.

To begin, we note that both full fences and Volatile pairs result in push orders in the formalism of [BP19]. That is, either a full fences or a volint edge implies a push edge. Our approach is to prove that when all program order accesses are push ordered then the semantics is SC. Thus, SC semantics follows as a corollary when all accesses are `volint`.

Recall from [BP19] that visibility order is acyclic. Intuitively, ordering induced by synchronization should not admit cycles.

▶ **Lemma 15** (Acyclic Visibility) [`coq/Truncate.v, vop_irreflex`]

If H is trace coherent then, acyclic($\xrightarrow{\mathtt{vvo}}$).

Next we show that the communication relation is not contradicted by visibility. Since the com relationship is composed from reads and coherence relationships, both of which encode the ordering of effects, we expect that visibility should not contradict such an ordering.

▶ **Lemma 16** (Communication Write Not-Visible) [`coq/SC/Volatile.v, coms_vo_contra`]

If H is trace coherent, all accesses are executed, $i_1 \xrightarrow{\mathsf{com}}{}^{*} i_2$ and i_2 is a write then $\neg(i_2 \xrightarrow{\mathtt{vvo}}{}^{+} i_1)$

Next we will establish that, when two pairs of `push` ordered accesses are connected by a possibly empty sequence of com edges, the first access of the first pair has been executed before the first access of the second pair. Intuitively, whenever there is a full fence between these two pairs of accesses then the order in which those fences executed must be consistent with the direction of the com relation.

▶ **Lemma 17** (Push Trace-ordered) [`coq/SC/Volatile.v, svo_comp_svo_to`]

If H is trace coherent, acyclic($\xrightarrow{\mathsf{co}}$), all accesses are executed, all accesses are `push` ordered, $i_1 \xrightarrow{\mathtt{push}} i_2 \xrightarrow{\mathsf{com}}{}^{*} i_3 \xrightarrow{\mathtt{push}} i_4$, then $i_1 \xrightarrow{\mathsf{to}} i_3$.

Proof. First, note that it is decidable whether i_3 is a write. We will begin by considering the case where it is a write. Since $\xrightarrow{\mathsf{to}}$ is total we consider each case for i_1 and i_3. First, if $i_1 \xrightarrow{\mathsf{to}} i_3$ we are done. Second, for $i_1 = i_3$ we will demonstrate a contradiction. By assumption we have $i_1 \xrightarrow{\mathtt{push}} i_2$, then by substitution we have $i_3 \xrightarrow{\mathtt{push}} i_2$. By the definition of push we have $i_3 \xrightarrow{\mathtt{vo}} i_2$.

By assumption we have $i_2 \xrightarrow{\text{com}}{}^* i_3$. By Lemma 16 we have $\neg(i_3 \xrightarrow{\text{vo}} i_2)$ and a contradiction. Finally, for $i_3 \xrightarrow{\text{to}} i_1$ we will also demonstrate a contradiction. By assumption we have both $i_1 \xrightarrow{\text{push}} i_2$ and $i_3 \xrightarrow{\text{push}} i_4$. Then by the definition of $\xrightarrow{\text{pushto}}$ and $i_3 \xrightarrow{\text{to}} i_1$ we have $i_3 \xrightarrow{\text{vo}} i_2$. As before we have $i_2 \xrightarrow{\text{com}}{}^* i_3$. By Lemma 16 we have $\neg(i_3 \xrightarrow{\text{vo}} i_2)$ and a contradiction.

Now consider the case where i_3 is not a write, then it must be a read and there exists some write i_w such that $i_w \xrightarrow{\text{rf}} i_3$. It can be shown that $i_2 \xrightarrow{\text{com}}{}^* i_w$. Thus we have $i_1 \xrightarrow{\text{push}} i_2 \xrightarrow{\text{com}}{}^* i_w \xrightarrow{\text{rf}} i_3 \xrightarrow{\text{push}} i_4$. Note that, because Lemma 16 applies to $i_2 \xrightarrow{\text{com}}{}^* i_w$ we have that $\neg(i_2 \xrightarrow{\text{vvo}}{}^+ i_w)$ and we must derive a contradiction by showing $i_w \xrightarrow{\text{vvo}}{}^+ i_2$. For $i_1 = i_3$ we have $i_3 \xrightarrow{\text{vvo}} i_2$ as before. Since $i_w \xrightarrow{\text{rf}} i_3$ by the definition of $\xrightarrow{\text{vvo}}$ we have $i_w \xrightarrow{\text{vvo}} i_3$ and $i_w \xrightarrow{\text{vvo}} i_2 \xrightarrow{\text{vvo}} i_3$ as required. For $i_3 \xrightarrow{\text{to}} i_1$ again we have that $i_3 \xrightarrow{\text{vvo}} i_2$ and in turn we have $i_w \xrightarrow{\text{vvo}} i_2 \xrightarrow{\text{vvo}} i_3$ as required. $\square$

Now we can demonstrate that when all accesses are push ordered the program semantics is SC. The key idea is that any cycle in the sc relation (i.e. a non-SC execution) will have at least one program order edge and at least one com edge. Thus we can show that the program order edge will appear twice in the cycle and then use Push Trace-ordered inductively to show that such a push order would have to execute before itself, thereby deriving a contradiction.

▶ THEOREM 25 (ALL PUSH SC)

[coq/SC/Volatile.v, push_sc]

If H is trace coherent, acyclic($\xrightarrow{\text{co}}$), all accesses are executed, and all accesses are push ordered then acyclic($\xrightarrow{\text{sc}}$).

Proof. We assume $i_1 \xrightarrow{\text{sc}} i_1$ and derive a contradiction. Observe that $i_1 \xrightarrow{\text{sc}} i_1$ must include at least one $\xrightarrow{\text{com}}$ edge because $\xrightarrow{\text{po}}$ is acyclic. Further observe that it must also include at least one program order edge because $\xrightarrow{\text{com}}$ is also acyclic. Thus there exists some access i_2 such that we can rearrange to obtain a sequence of program order and communication edges, $i_2(\xrightarrow{\text{po}}\xrightarrow{\text{com}}{}^+)^+ i_2$. We proceed by induction on the length of this sequence. In the base case there exists some i_3 such that $i_2 \xrightarrow{\text{po}} i_3 \xrightarrow{\text{com}}{}^+ i_2$ which wraps around to give $i_2 \xrightarrow{\text{po}} i_3 \xrightarrow{\text{com}}{}^+ i_2 \xrightarrow{\text{po}} i_3$. Then Push Trace-ordered applies to give $i_2 \xrightarrow{\text{to}} i_2$, but this is a contradiction since the trace order is total. In the inductive case we use the same argument and connect the trace order from the inductive hypothesis to give a contradiction. □

From Theorem 25 we can derive two corollaries. The first shows that when all accesses are Volatile the semantics is SC. The second shows that when all accesses have full fences between them, represented by spush in the model, the semantics is SC. The structure of the model and our definition for Volatile accesses shines through here as both results follow directly from a single result about the behavior of full fences.

▶ Corollary 5 (All Volatile SC)

[coq/SC/Volatile.v, volatile_sc]

If H is trace coherent, acyclic($\xrightarrow{\text{co}}$), all accesses are executed, and all accesses are Volatile mode accesses then acyclic($\xrightarrow{\text{sc}}$).

Proof. If all accesses are volatile then any two program order accesses are push ordered and we can appeal to Theorem 25. □

▶ Corollary 6 (All Specified Push SC)

[coq/SC/Volatile.v, spush_sc]

If H is trace coherent, acyclic($\xrightarrow{\text{co}}$), all accesses are executed, and all program ordered (po) accesses are ordered by specified push order (spush) then acyclic($\xrightarrow{\text{sc}}$).

Proof. If any two program order accesses have a specified push order then they are similarly push ordered and we can again appeal to Theorem 25. □

APPENDIX I

The Standard DRF-SC Theorem

▶ THEOREM 22 (DRF-SC)
Given a program P, if all its SC-consistent executions are data-race-free or only have volatile-races, then the set of all JAM21-consistent executions of P coincide with the set of SC-consistent executions.

Proof. Let P be a program, and suppose all its SC-consistent executions only has Volatile-races. We want to show that P has no weak behavior. Toward contradiction, let's assume there exists an execution G of P such that G is JAM21-consistent but not SC-consistent.

▶ DEFINITION 28
An execution G' is called a *prefix* of an execution G if G' is obtained by restricting G to a set of events E such that:

1. the set of initialization events $E_0 \in E$
2. for any event $b \in E$, if there is $a \xrightarrow{\text{po}} b$ or $a \xrightarrow{\texttt{rf}} b$ in G, then $a \in E$. (Closed with respect to $(G.\text{po} \cup G.\texttt{rf})$)

Claim 1 Any prefix of a JAM21-consistent execution is JAM21-consistent.

Claim 2 Any prefix of an SC-consistent execution is SC-consistent.

Proof. The above two claims are true because a prefix consists of a subset of edges and events of the original execution. If there is a cycle that violates the requirements of the memory models, then the same cycle is present in the original execution graph. Therefore, since we assumed G is JAM21-consistent, any prefix of G is also JAM21-consistent. □

Notations For a set of events E, let $\Pi(E)$ denote the set of all pairs $\langle a, b\rangle \in E \times E$ of conflicting events, such that $\{G.AccessMode(a), G.AccessMode(b)\} \neq \{\text{Volatile}\}$ and $\langle a, b\rangle$, $\langle b, a\rangle \notin (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.

$$\Pi(E) = \{\langle a, b\rangle \in E \times E \mid \{G.AccessMode(a), G.AccessMode(b)\} \neq \{\text{Volatile}\},\\ \langle a, b\rangle, \langle b, a\rangle \notin (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+\}$$

Let $a_1, ..., a_n$ be an enumeration of events ordered by trace orders (recall that trace order is a total order among the events in an execution that is compatible with $(G.\text{po} \cup G.\texttt{rf})^+$).

Let E_i denotes the subset of events $E_0 \cup \{a_1, ..., a_i\}$ and G_i be the execution restrict to E_i. This is easy to see that each G_i is a prefix to G because the trace order is compatible with $(G.\text{po} \cup G.\texttt{rf})^+$. Therefore, G_i is also JAM21-consistent by **Claim 1**.

Claim 3 For every $1 \leq i \leq n$, if $\Pi(E_i) = \emptyset$, then G_i is SC-consistent.

Proof. Suppose that $\Pi(E_i) = \emptyset$. Then, for every conflicting pair $\langle a, b\rangle$, either $G.AccessMode(a) = G.AccessMode(b) = \text{Volatile}$ or $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$ or $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.

1. For any $a \xrightarrow{\texttt{rf}} b$ in G_i.

   - If $G.AccessMode(a) = G.AccessMode(b) = \text{Volatile}$, then $\langle a, b\rangle \in G.\texttt{rf} \restriction_{\text{Volatile}}$.
   - If $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.
   - If $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then there is a $(\text{po} \cup \texttt{rf})+$ cycle between a and b. By the NO-THIN-AIR requirement, G_i is not JAM21-consistent. Contradicting to our previous assumption. Therefore it is impossible to have $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.

Thus, $G_i.\texttt{rf} \subseteq (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$

2. For any $a \xrightarrow{\texttt{co}} b$ in G_i,

   - If $G.AccessMode(a) = G.AccessMode(b) =$ Volatile, then $\langle a, b\rangle \in G.\texttt{co} \restriction_{\text{Volatile}}$.
   - If $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.
   - If $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then the the domains of the $G.\texttt{rf}$ on the path from b to a has access mode equal to Volatile which includes Release semantics. Similarly, the ranges of the $G.\texttt{rf}$ have access mode equal to Volatile which includes Acquire semantics. Therefore, po $\subseteq$ $\texttt{ra}$ on this path. That is, we have $\langle b, a\rangle \in (G.\texttt{ra} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+ \subseteq G.\texttt{vvo}^+$. By $\texttt{coww}$, we have $\langle b, a\rangle \in G.\texttt{co}$. With $a \xrightarrow{\texttt{co}} b$, we now have a $\texttt{co}$ cycle, contradicting to the earlier assumption that G_i is JAM21-consistent. Therefore, it is impossible that $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.

   Thus, we have $G_i.\texttt{co} \subseteq (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+ \cup G.\texttt{co} \restriction_{\text{Volatile}}$.

3. For any $a \xrightarrow{\texttt{fr}} b$ in G_i,

   - If $G.AccessMode(a) = G.AccessMode(b) =$ Volatile, then $\langle a, b\rangle \in G.\texttt{fr} \restriction_{\text{Volatile}}$.
   - If $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then $\langle a, b\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.
   - If $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$, then the the domains of the $G.\texttt{rf}$ on the path from b to a has access mode equal to Volatile which includes Release semantics. Similarly, the ranges of the $G.\texttt{rf}$ have access mode equal to Volatile which includes Acquire semantics. Therefore, po $\subseteq$ $\texttt{ra}$ on this path. That is, we have $\langle b, a\rangle \in (G.\texttt{ra} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+ \subseteq G.\texttt{vvo}^+$. Expanding the definition of $\texttt{fr}$, there exists a write event i such that $\langle i, a\rangle \in G_i.\texttt{rf}$ and $\langle i, b\rangle \in G_i.\texttt{co}$. By $\texttt{cowr}$, we have $\langle b, i\rangle \in G_i.\texttt{co}$. Now we have a $\texttt{co}$ cycle, contradicting with the earlier assumption that G_i is JAM21-consistent. Therefore, it is impossible that $\langle b, a\rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$.

   Thus, we have $G_i.\texttt{fr} \subseteq (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+ \cup G.\texttt{fr} \restriction_{\text{Volatile}}$.

Therefore, we have $G_i.\text{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co} \subseteq G.\text{po} \cup \texttt{rf} \restriction_{\text{Volatile}} \cup \texttt{fr} \restriction_{\text{Volatile}} \cup \texttt{co} \restriction_{\text{Volatile}}$. Since G_i is JAM21-consistent, any prefix of G_i is JAM21-consistent as well (by **Claim 1**). By our previous lemma that *Volatile* $\Rightarrow$ *SC*, any prefix of G_i with all events marked as Volatile are SC-consistent. If G_i is not SC-consistent and there is a cycle of $G.\text{po} \cup \texttt{rf} \restriction_{\text{Volatile}} \cup \texttt{fr} \restriction_{\text{Volatile}} \cup \texttt{co} \restriction_{\text{Volatile}}$, then there exists a prefix of G_i containing all the events in this cycle and hence not SC-consistent. This contradicts with our previous assumption. Thus, G_i is SC-consistent. □

Now, continuing our proof, since G is not SC-consistent, we know that $\Pi(E) \neq \emptyset$.

Convention In the rest of the proof, we treat an Read-modify-write (RMW) event as two separate events ordered by the `rmw` order. That is, each RMW event in G consists of two events i_1 and i_2 such that $\langle i_1, i_2 \rangle \in G.\texttt{rmw}$, where $i_1 \in G.\text{Rds}$ and $i_2 \in G.\text{Wrts}$.

Let $k = min\{i \mid \Pi(E_i) \neq \emptyset\}$. Then it is clear that G_{k-1} is the maximum SC-consistent prefix of G. That is, $\Pi(E_{k-1}) = \emptyset$ and G_{k-1} is SC-consistent. We also have $\Pi(E_k) \neq \emptyset$ and G_k is not SC-consistent. That is, there exists $j < k$ such that $\{G.AccessMode(a_j), G.AccessMode(a_k)\} \neq \{\text{Volatile}\}$, and $\langle a_j, a_k \rangle$, $\langle a_k, a_j \rangle \notin G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}}$.

Claim 4 Let $B = \{b \mid \langle b, a_k \rangle \in G_k.\text{po}\}$, then $\langle a_j, b \rangle \notin (G.\text{po} \cup G.\texttt{rf})^+$.

Proof. Since $\Pi(E_{k-1}) = \emptyset$, we have $G_{k-1}.\texttt{rf} \subseteq (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$. If $\langle a_j, b \rangle \in (G.\text{po} \cup G.\texttt{rf})^+$, then we have $\langle a_j, b \rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})^+$. From that we have $\langle a_j, a_k \rangle \in (G.\text{po} \cup G.\texttt{rf} \restriction_{\text{Volatile}})+$ since $\langle b, a_k \rangle \in G.\text{po}$, which contradicts to our previous assumption. Therefore, $\langle a_j, b \rangle \notin (G.\text{po} \cup G.\texttt{rf})^+$. □

1. $a_k \in G.\text{Wrts}$.

Claim 5 $\langle a_j, a_k \rangle$ forms a data race.

Proof. Since G_k is closed under $(G.\mathsf{po} \cup G.\mathtt{rf})$ and a_k is the last event in the total trace order in G_k, there is no outgoing edge from a_k. Therefore, $\langle a_k, a_j \rangle \notin (G.\mathsf{po} \cup G.\mathtt{rf})^+$. In addition, Since a_k is a write, we cannot have $\langle a_j, a_k \rangle \in G.\mathtt{rf}$. Therefore, if $\langle a_j, a_k \rangle \in G.\mathtt{hb}$, then it would imply that $\langle a_j, b \rangle \in (G.\mathsf{po} \cup G.\mathtt{rf})^+$ for some b such that $\langle b, a_k \rangle \in G.\mathsf{po}$, which contradicts with **Claim 4**. Therefore, $\langle a_j, a_k \rangle \notin (G.\mathsf{po} \cup G.\mathtt{rf})^+$ and $\langle a_j, a_k \rangle$ forms a race in G_k. □

Claim 6 G_k is not SC-consistent.

Proof. Given that $\langle a_j, a_k \rangle$ forms a data race in G_k and $\{G.AccessMode(a_j), G.AccessMode(a_k)\} \neq \{\mathsf{Volatile}\}$, this follows from the assumption. □

Claim 7 There does not exists a read event b such that $\langle b, a_k \rangle \in G.\mathtt{rmw}$.

Proof. Suppose toward conradiction that there is b such $\langle b, a_k \rangle \in G.\mathtt{rmw}$. Since G_k is not SC-consistent, there is cycle of $(G_k.\mathsf{po} \cup G_k.\mathtt{rf} \cup G_k.\mathtt{fr} \cup G_k.\mathtt{co})^+$. Since G_{k-1} is SC-consistent, it must be that a_k is part of the cycle in G_k. That is, there is a $(G_k.\mathsf{po} \cup G_k.\mathtt{rf} \cup G_k.\mathtt{fr} \cup G_k.\mathtt{co})$ edge from a_k. Additionally, it cannot be a po or $\mathtt{rf}$ because G_k is a closed prefix of G. Since a_k is a write, it cannot be $\mathtt{fr}$ either. Therefore, there is some c in G_{k-1} such that $\langle a_k, c \rangle \in G.\mathtt{co}$ and $\langle c, a_k \rangle \in (G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{fr} \cup G.\mathtt{co})^+$. Extending the edges, we have $\langle c, d \rangle \in (G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{fr} \cup G.\mathtt{co})^*$, and $\langle d, a_k \rangle \in (G.\mathtt{co} \cup G.\mathtt{fr} \cup G.\mathsf{po})$. We analyze each case below.

- $\langle d, a_k \rangle \in G.\mathtt{co}$. Then we know that $c, d \in G_{k-1}$ are writes to the same location. Since G_{k-1} is SC-consistent, and $\langle c, d \rangle \in (G_{k-1}.\mathsf{po} \cup G_{k-1}.\mathtt{rf} \cup G_{k-1}.\mathtt{fr} \cup G_{k-1}.\mathtt{co})^*$, we have $\langle c, d \rangle \in G_{k-1}.\mathtt{co}^*$. If $c = d$, then we have a $G_k.\mathtt{co}$ cycle between c and a_k, making G_k not JAM21-consistent. Therefore, we have $\langle c, d \rangle \in G_{k-1}.\mathtt{co}$. But we also have $\langle d, a_k \rangle, \langle a_k, c \rangle \in G_k.\mathtt{co}$. Together, they yield a $\mathtt{co}$ cycle, contradicting with our eariler assumption that G_k is JAM21-consistent.

- $\langle d, a_k \rangle \in G.\mathtt{fr}$. Then we know d is a read. Given that $\langle a_k, c \rangle \in G.\mathtt{co}$, we can infer that $\langle d, c \rangle \in G_{k-1}.\mathtt{fr}$. But we also have $\langle c, d \rangle \in (G_{k-1}.\mathrm{po} \cup G_{k-1}.\mathtt{rf} \cup G_{k-1}.\mathtt{fr} \cup G_{k-1}.\mathtt{co})^*$, which forms a cycle between c and d, contradicting to the assumption that G_{k-1} is SC-consistent.
- $\langle d, a_k \rangle \in G.\mathrm{po}$. Then we have $\langle d, b \rangle \in G_{k-1}.\mathrm{po}^?$. Since $\langle b, a_k \rangle \in G_k.\mathtt{rmw}$, by `cormwexcl`, we know that there exists some e such that $\langle e, b \rangle \in G_{k-1}.\mathtt{rf}$ and $\langle e, c \rangle \in G_{k-1}.\mathtt{co}$. Therefore, $\langle b, c \rangle \in G_{k-1}.\mathtt{fr}$. However, we also have $\langle c, d \rangle \in (G_{k-1}.\mathrm{po} \cup G_{k-1}.\mathtt{rf} \cup G_{k-1}.\mathtt{fr} \cup G_{k-1}.\mathtt{co})^*$ and $\langle d, b \rangle \in G_{k-1}.\mathrm{po}^?$. Now we have a cycle contradicting to the assumption that G_{k-1} is SC-consistent.

□

Now let G'_k be a transformation of G_k such that for all $\langle a_k, c \rangle \in G_k.\mathtt{co}$, we have $\langle c, a_k \rangle \in G'_k.\mathtt{co}$. Since there is no b such that $\langle b, a_k \rangle \in G.\mathtt{rmw}$, transforming the $\mathtt{co}$ edges in G_k in this way won't affect any other $\mathtt{fr}$ or $\mathtt{rf}$ edges in G_k. As a result, G'_k is the same as G_k except for the $\mathtt{co}$ edges. Moreover, G'_k is SC-consistent because we have established that the only way to form a cycle in G_k is to have $\langle a_k, c \rangle \in G_k.\mathtt{co}$ for some c and G'_k essentially breaks the cycle by flipping the $\mathtt{co}$ arrows. However, we still have $\langle a_j, a_k \rangle$ forming a race in G'_k and $\{G'_k.AccessMode(a_j), G'_k.AccessMode(a_k)\} \neq \{\mathsf{Volatile}\}$.

Claim 8 All prefixes of SC-consistent executions of P are data race free.

Proof. This follows from the fact that prefixes are closed under $\mathrm{po} \cup \mathtt{rf}$. □

Claim 9 G'_k is a prefix of some SC-consistent execution of P.

Proof. Since G'_k is SC-consistent and closed under $\mathrm{po} \cup \mathtt{rf}$, we can construct an SC-consistent execution G' such that for all event $i \in G'.E \backslash G'_k.E$ and $e \in G'_k.E$, we have $\langle e, i \rangle \in G'.\mathtt{to}$ (trace order). □

It is clear that the fact G'_k has a data race $\langle a_j, a_k\rangle$ contradicts with **Claim 8** and **Claim 9**.

2. $a_k \in G$.Rds.

   Then we know that a_j is a write.

   Let $E = \{a \in G.E \,|\, \langle a, a_k\rangle \in (G.\text{po} \cup G_{k-1}.\texttt{rf})^*\} \cup \{a \in G.E \,|\, \langle a, a_j\rangle \in (G.\text{po} \cup G_{k-1}.\texttt{rf})^*\}$. Note that there does not exist any $a \in E$ such that $\langle a_j, a\rangle \in (G.\text{po} \cup G_{k-1}.\texttt{rf})^*$ and $\langle a, a_k\rangle \in (G.\text{po} \cup G_{k-1}.\texttt{rf})^*$ since G_{k-1} is closed under $(G.\text{po} \cup G.\texttt{rf})$.

   Let G' be the restriction of G_k to the events in E.

   Claim 10 $\langle a_j, a_k\rangle$ forms a data race in G'

   Proof. Since a_k is a read event, there is no out-going $\texttt{rf}$ edge from a_k. Because G' itself is closed under $G.\text{po} \cup G.\texttt{rf}$, we have $\langle a_k, a_j\rangle \notin (G'.\text{po} \cup G'.\texttt{rf})^+$. In addition, there is no out-going edge from a_j in G', so $\langle a_j, a_k\rangle \notin (G'.\text{po} \cup G'.\texttt{rf})^+$. □

   Now, we want to construct an SC-consistent execution that contains the race $\langle a_j, a_k\rangle$ to show a contradiction.

   Let x be the location a_k accesses and c be the last write to x according to $G'.\texttt{co}$.

   - $c \neq a_j$. Let d be the write to x such that $\langle d, a_k\rangle \in G'.\texttt{rf}$. We transform G' so that a_k reads from c instead of d. That is, we remove $\langle d, a_k\rangle$ from $G'.\texttt{rf}$, change the value of a_k to the value of c, and add $\langle c, a_k\rangle$ to $G'.\texttt{rf}$. The immediate consequence of this transformation is, for all e such that $\langle d, e\rangle \in G'.\texttt{co}$, the $\texttt{fr}$ edge from a_k to e are also removed. Since a_k is a read event, there cannot be $\texttt{rf}$ or $\texttt{co}$ edge going out from a_k. As a result, we cannot form a $(\text{po}\cup\texttt{rf}\cup\texttt{co}\cup\texttt{fr})^+$ cycle with a_k and the resulting execution is SC-consistent. Then the fact that $\langle a_j, a_k\rangle$ is a data race gives us a contradiction.
   - $c = a_j$. That is, a_k forms a race with the last write to the same location. Let d be the write to x such that $\langle a, a_k\rangle \in G'.\texttt{rf}$. We transform G' so that a_k reads from the write that immediately $\texttt{co}$ ordered before c. Let e be that write. We remove $\langle d, a_k\rangle$

from $G'.\texttt{rf}$, change the value of a_k to the value of e, and add $\langle e, a_k \rangle$ to $G'.\texttt{rf}$. Since $\langle e, c \rangle \in G'.\texttt{co}$, we have $\langle a_k, c \rangle \in G'.\texttt{fr}$. However, since $c = a_j$ and $\langle a_j, a_k \rangle$ forms a race, $\langle c, a_k \rangle \notin (G'.\mathsf{po} \cup G'.\texttt{rf})^+$. That is, there is no path from c to a_k in G'. As a result, we cannot form any cycle with the added $\texttt{fr}$ edge from a_k and the execution after the transformation is SC-consistent. Since we still have $\langle c, a_k \rangle$ forming a race, we now have a contradiction.

$\square$

APPENDIX J

DRF-SC for Execution Graphs

▶ THEOREM 23 (EXECUTION-DRF)
Any JAM21-consistent execution that is data race free or only has volatile-races is SC-consistent.

Proof. Let P be a program. We first consider the case without any Volatile-race. That is, all conflicting pairs of accesses are ordered by the happens-before ($\texttt{hb}$) order in some SC-consistent execution of P. We prove that there does not exist any JAM21-consistent execution of P that is not SC-consistent. Suppose toward a contradiction that there exists an execution G of P such that G is JAM21-consistent, race-free, but not SC-consistent. That is, G has a $(\mathsf{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})+$ cycle. In addition, we assume that $\texttt{co-jam} \subseteq \texttt{co}$.

First note that each of the communication edges in the cycle of G are also pairs of conflicting accesses. Indeed, they are defined between accesses to the same location and at least one of the accesses is a write event. Then, by our assumption that G data-race-free, they are also ordered by the $\texttt{hb}$ order. In addition, for all conflicting accesses i_1 and i_2 in G:

- $i_1 \xrightarrow{\texttt{rf}} i_2 \Rightarrow i_1 \xrightarrow{\texttt{hb}} i_2$
- $i_1 \xrightarrow{\texttt{fr}} i_2 \Rightarrow i_1 \xrightarrow{\texttt{hb}} i_2$
- $i_1 \xrightarrow{\texttt{co}} i_2 \Rightarrow i_1 \xrightarrow{\texttt{hb}} i_2$

because the other direction can immediately lead to contradictions.

Then the cycle in G a cycle of $(\mathsf{po} \cup \texttt{hb})+$. We expend the definition of $\texttt{hb}$, we have a cycle of $(\mathsf{po} \cup (\mathsf{po} \cup \texttt{sw})+)+$, which simplifies to a $(\mathsf{po} \cup \texttt{sw})+$ cycle.

By the definition of sw order, the domain of each sync edge is a Release mode write (or a release fence followed by a write) and the range of sw is an Acquire mode read (or a read followed by an acquire fence). Therefore, we know that the "head" of each thread in this cycle is an Acquire read and the "end" of each thread in this cycle is a Release write. By the semantics of Release-Acquire mode, all of the po order to a Release write and all of the po order from an Acquire read is preserved and captured in the ra order, which is a subset of vo. In addition, $\texttt{sw} \subseteq \texttt{vo}$. As a result, the cycle in G is actually a vo cycle. However, we assumed that G is JAM21-consistent and by the previous lemma by Bender et al. [BP19], the vo order is acyclic in all JAM21-consistent executions. Thus a contradiction.

We now consider the case where there are Volatile-races in the execution. We prove this by incrementally inserting a pair of Volatile-race into an execution that is data-race-free and JAM21-consistent and prove that it does not introduce any weak behavior to the execution. Let G' be such an execution. As we just have shown, G' is SC-consistent. We would like to insert a pair of Volatile-race $\langle a, b\rangle$ into G'. Let T_1 be the thread where a is inserted and T_2 be the thread where b is inserted. By definition of data race, $T_1 \neq T_2$. We have three possible cases:

- T_1 and T_2 are not connected by any $(\texttt{po} \cup \texttt{sw})+$ edge before inserting $\langle a, b\rangle$ in G'. That is, there does not exist any $(\texttt{po} \cup \texttt{sw})+$ path from T_1 to T_2. Then inserting $\langle a, b\rangle$ into the execution cannot form any cycle in $(\texttt{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})+$ since it can only add at most one edge to the graph.

- There is a $(\texttt{po} \cup \texttt{sw})+$ path from T_1 to T_2. First note that G' before inserting $\langle a, b\rangle$ satisfies that, if two threads are connected, then they must be connected by at least an sw edge. This implies there is a release write W^{Rel} on T_1, an acquire read R^{Acq} on T_2, and $W^{Rel} \xrightarrow{\texttt{sw}} \xrightarrow{(\texttt{po}\,|\,\texttt{sw})*} R^{Acq}$. Due to this structure, the only way to insert $\langle a, b\rangle$ and build a cycle of $(\texttt{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})+$ is to insert a before W^{Rel} and insert b after R^{Acq}. Note that this implies $a \xrightarrow{\texttt{ra}} W^{Rel} \xrightarrow{\texttt{sw}} \xrightarrow{(\texttt{po}\,|\,\texttt{sw})*} R^{Acq} \xrightarrow{\texttt{ra}} b$. So depending on what type of access a and b are, we have three cases:
  - a is a Volatile write and b is a Volatile read and $b \xrightarrow{\texttt{fr}} a$. By definition of fr, there exists

a write i such that $i \xrightarrow{\texttt{rf}} b$ and $i \xrightarrow{\texttt{co}} a$. But $a \xrightarrow{\texttt{ra}} W^{Rel} \xrightarrow{\texttt{sw}} \xrightarrow{(\texttt{po} \mid \texttt{sw})*} R^{Acq} \xrightarrow{\texttt{ra}} b$ implies that $a \xrightarrow{\texttt{vo}} b$, and by the `cowr` coherence rule, we have $a \xrightarrow{\texttt{co}} i$. Now we have a `co` cycle, contradicting to our assumption that G'' is JAM21-consistency.

  - a is a Volatile read and b is a Volatile write and $b \xrightarrow{\texttt{rf}} a$. Again, we have $a \xrightarrow{\texttt{vo}} b$ and $b \xrightarrow{\texttt{rf}} a$. Since $\texttt{rf} \subseteq \texttt{vo}$, we get a `vo` cycle, contradicting to our previous assumption of JAM21-consistency.
  - a is a Volatile write and b is a Volatile write and $b \xrightarrow{\texttt{co}} a$. Again, we have $a \xrightarrow{\texttt{vo}} b$ and $b \xrightarrow{\texttt{co}} a$. By `coww`, we have $a \xrightarrow{\texttt{co}} b$ and $b \xrightarrow{\texttt{co}} a$, a coherence cycle contradicting to our previous assumption of JAM21-consistency.

- There is a $(\texttt{po} \cup \texttt{sw})+$ path from T_2 to T_1. Symmetrical to the previous case.

$\square$

APPENDIX K

Experimental Validation of JAM21

In this chapter, we explain our implementation of Java architecture for HERD7 [AMT14] and experimental results with the JAM21 model. The source code of our Java architecture implementation will become available for artifact evaluation.

K.1 Methods supported by Java Architecture for Herd7

The list of supported methods in our implementation of Java in HERD7 [AMT14] can be found in Fig. K.1. We provide a description for each one of the method and its corresponding action in Herd7.

K.2 Experimental Results

In this section, we show the experimental results of running the same set of litmus tests as in JAM19 and compare their outcomes. Three types of results can be yielded by Herd7 at the end, `Always`, `Sometimes`, and `Never`. `Always` and `Sometimes` means the behavior specified in the litmus test is allowed, whereas `Never` means it is forbidden.

Fig. K.2 shows the experimental results of running volatile-non-sc.4 example and its 5-thread version with the JAM19 and the JAM21 model. As we expected, the update to the JAM21 model fixes the issue we addressed earlier in the paper and the executions changed from `Sometimes` to `Forbidden`.

Method	**Memory Action**	**Description**
`getM()`	R(M)	Read operation with access mode specified by `M`, where `M` can be omitted (Plain mode), `Opaque`, `Acquire`, or `Volatile`.
`setM(val)`	W(M)	Write operation with access mode specified by `M`, where `M` can be omitted (Plain mode), `Opaque`, `Acquire`, or `Volatile`, the value written is specified by `val`, which can be either an integer or a local variable.
`compareAndExchangeM(expect, dest)`	RMW(M)	An atomic compare and update operation with access mode specified by M, where `M` can be omitted (Volatile mode), `Acquire`, or `Release`.
`getAndOpM(val)`	RMW(M)	A numeric or bitwise atomic update operation with modifying operation specified by `Op` and access mode specified by `M`, where `Op` can be `Add`, `And`, `Or`, or `Xor`; and `M` can be omitted, `Acquire`, or `Release`.
`fullFence()`	F(Volatile)	A full fence.
`releaseFence()`	F(Release)	A release fence.
`acquireFence()`	F(Acquire)	An acquire fence.

Figure K.1: Methods supported by the Java Architecture

Name	Jam19	Jam21
volatile-non-sc.4	`Sometimes`	`Never`
volatile-non-sc.5	`Sometimes`	`Never`

Figure K.2: volatile-non-sc Experimental Results

Fig. K.3 shows the rest of the experimental results in details. Note that not all litmus tests used for Jam19 [BP19] are translatable to Java. We marked those non-translatable tests as “N/A” in the tables (since Java does not have the notion of address dependency). The result agrees with our expectation that most of the litmus tests yield the same results as Jam19, except those that are related to the inconsistency issue (highlighted using bold font). We discuss each of the exceptions.

The execution graphs of IRIW-acq-sc are shown in Fig. K.4. Our experimental results show that this execution is forbidden under Jam19 but is allowed under Jam21. Under the Jam19 model, because the definition of `volint` includes orders from any instruction to a Volatile read program ordered after the instruction, we have $\texttt{(c)}\xrightarrow{\texttt{volint}}\texttt{(d)}$ and $\texttt{(e)}\xrightarrow{\texttt{volint}}\texttt{(f)}$. Between the two threads (P3 and P4), there is the visibility order $\texttt{(c)}\xrightarrow{\texttt{vo}}\texttt{(f)}$, or $\texttt{(e)}\xrightarrow{\texttt{vo}}\texttt{(d)}$. Both cases can produce the contradictory result that one of the threads observes the non-initialization write before the initialization write, i.e., a coherence cycle. Therefore, this execution is forbidden under the Jam19 model. In Jam21, the two `volint` orders are no longer present in the execution graph because the new definition of `volint` requires both of the memory accesses to be Volatile. As a result, the execution becomes allowed under the Jam21 model. To see why allowing this execution is an improvement, note that the Jam19 model only captures the "leading fence" convention that `fullFence()` are inserted before Volatile accesses. On the other hand, if the compiler follows the "trailing fence" convention, there would not be `fullFence()`s in `P3` and `P4`. In that case, the execution is allowed. In order to accommodate both conventions, the Jam21 model relaxes to allow this execution.

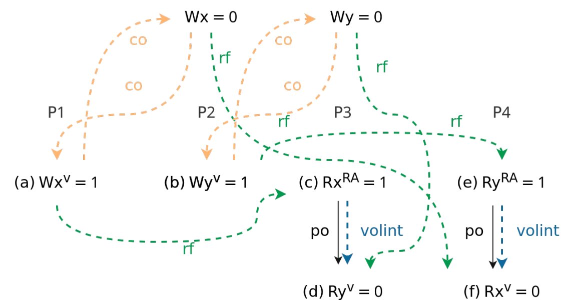
The execution graph of Z6.U are shown in Fig. K.5. Due to the original problematic encoding of Volatile writes, `volint` includes orders from Volatile writes to any program ordered later memory accesses. Therefore, $\texttt{(a)}\xrightarrow{\texttt{volint}}\texttt{(b)}$, $\texttt{(c)}\xrightarrow{\texttt{volint}}\texttt{(d)}$, and $\texttt{(e)}\xrightarrow{\texttt{volint}}\texttt{(f)}$. Similar to the

Name	JAM19	JAM21
WRC+addrs	Never	N/A
LB+data+data-wsi	Never	N/A
W+RR	Never	Never
totalco	Never	Never
PPOCA	Sometimes	N/A
IRIW	Sometimes	Sometimes
IRIW+addrs	Sometimes	N/A
IRIW+poaas+LL	Sometimes	Sometimes
IRIW+poaps+LL	Sometimes	Sometimes
MP+dmb.sy+addr-ws-rf-addr	Sometimes	N/A
WW+RR+WW+RR+wsilp+poaa+wsilp+poaa	Sometimes	Sometimes
LB	Never	Never

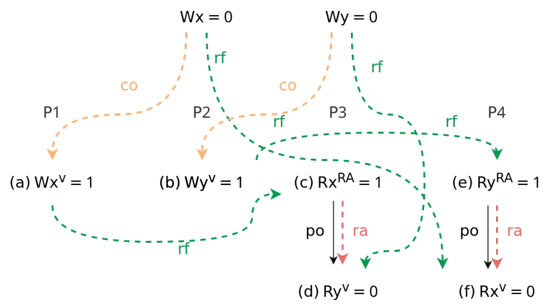
Name	JAM19	JAM21
a1	Sometimes	Sometimes
a1_reorder	Sometimes	Sometimes
a3	Sometimes	Sometimes
a3_reorder	Sometimes	Sometimes
a3v2	Sometimes	Sometimes
a4	Never	Never
a4_reorder	Sometimes	Sometimes
arfna	Never	Never
arfna_transformed	Never	Never
b	Never	Never
b_reorder	Sometimes	Sometimes
c	Never	Never
c_p	Never	Never
c_p_reorder	Never	Never
c_pq	Never	Never
c_pq_reorder	Never	Never
c_q	Never	Never
c_q_reorder	Never	Never
c_reorder	Never	Never
cyc	Never	Never
cyc_na	Sometimes	Sometimes
fig1	Always	Always
fig6	timed out	time out
fig6_translated	timed out	time out
lb	Never	Never
linearisation	Never	Never
linearisation2	Never	Never
roachmotel	Never	Never
roachmotel2	Never	Never
rseq_weak	Sometimes	Sometimes
rseq_weak2	Always	Always
seq	Never	Never
seq2	Never	Never
strengthen	Never	Never
strengthen2	Never	Never

Name	JAM19	JAM21
2+2W	Never	Never
IRIW-acq-sc	**Never**	**Sometimes**
RWC+syncs	Never	Never
W+RWC	Never	Never
Z6.U	**Never**	**Sometimes**
IRIW-sc-rlx-acq	**Never**	**Sometimes**
cppmem_iriw_relacq	Sometimes	Sometimes
cppmem_sc_atomics	Never	Never
iriw_sc	Never	Never
mp_fences	Never	Never
mp_relacq	Never	Never
mp_relacq_rs	Sometimes	Sometimes
mp_relaxed	Sometimes	Sometimes
mp_sc	Never	Never
4.SB	Sometimes	Sometimes
6.SB	timeout	timeout
6.SB+prefetch	timeout	timeout
CoRWR	Never	Never
SB+SC	Sometimes	Sometimes
SB+mfences	Never	Never
SB+rfi-pos	Sometimes	Sometimes
SB	Sometimes	Sometimes
X000	Sometimes	Sometimes
X001	Sometimes	Sometimes
X002	Sometimes	Sometimes
X003	Sometimes	Sometimes
X004	Sometimes	Sometimes
X005	Sometimes	Sometimes
X006	Sometimes	Sometimes
iriw-internal	Sometimes	Sometimes
iriw	Sometimes	Sometimes
podrw000	Sometimes	Sometimes
podrw001	Sometimes	Sometimes
x86-2+2W	Sometimes	Sometimes

Figure K.3: Litmus Test Comparisons

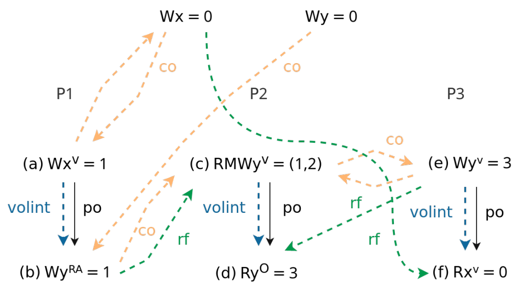


(a) Before: Forbidden

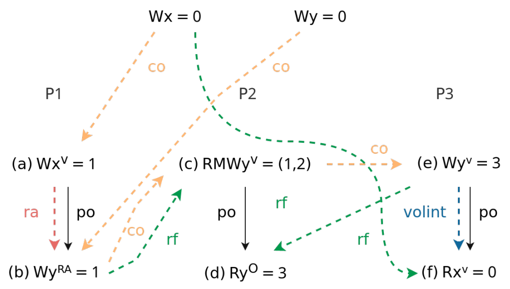


(b) After: Allowed

Figure K.4: IRIW-acq-sc

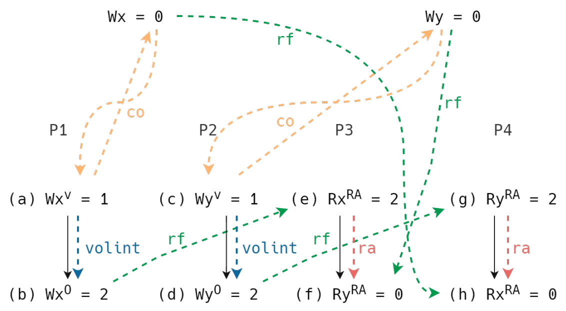


(a) Before: Forbidden

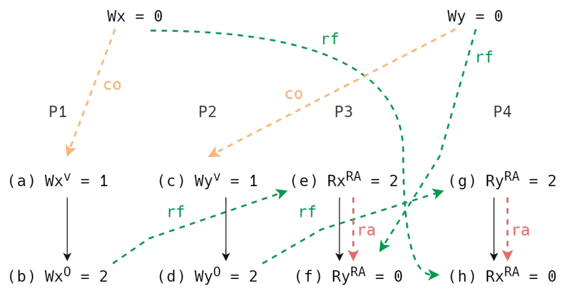


(b) After: Allowed

Figure K.5: Z6.U



(a) Before: Forbidden



(b) After: Allowed

Figure K.6: IRIW-seq-rlx

previous example, there are two possible visibility orders, `(a)`$\xrightarrow{vo}$`(f)` or `(e)`$\xrightarrow{vo}$`(b)`. The former case leads to the derivation of `(a)`$\xrightarrow{co}$`(Wx=0)`, which contradicts the assumption that all initialization writes to variable `x` are ordered before all non-initialization writes to `x`. The latter case leads to a contradiction as well. Because `(d)` reads the value written by `(e)` and `(c)`$\xrightarrow{volint}$`(d)`, we can infer that `(c)`$\xrightarrow{co}$`(e)`. If `(e)`$\xrightarrow{vo}$`(b)`, then `(e)`$\xrightarrow{vo}$`(c)`. By the coww rule, `(e)`$\xrightarrow{co}$`(c)`. This leads to a coherence `co` cycle between `(c)` and `(e)`. The JAM21 model relaxes the `volint` edges in `P1` and `P2` in order to accommodate both the leading fence convention and the trailing fence convention. If the compiler follows the convention of inserting `fullFence()` before the Volatile accesses, there is only `(a)`$\xrightarrow{ra}$`(b)` in `P1` and no synchronization between the two instructions in `P2`. Thus this execution is allowed under the JAM21 model.

Lastly, the execution graphs of IRIW-seq-rlx are shown in Fig. K.6. Originally, due to the old encoding of Volatile writes, `(a)`$\xrightarrow{volint}$`(b)` and `(c)`$\xrightarrow{volint}$`(d)`. Two possible visibility orders can be inferred, either `(a)`$\xrightarrow{vo}$`(d)` or `(c)`$\xrightarrow{vo}$`(b)`. The former case leads to the conclusion that `(a)`$\xrightarrow{co}$`Wx=0` because `(a)`$\xrightarrow{vo}$`(d)`$\xrightarrow{rf}$`(g)`$\xrightarrow{ra}$`(h)` and `(Wx=0)`$\xrightarrow{rf}$`(h)`. Similarly, the latter case leads to the conclusion that `(c)`$\xrightarrow{co}$`(Wy=0)` because `(c)`$\xrightarrow{vo}$`(b)`$\xrightarrow{rf}$`(e)`$\xrightarrow{ra}$`(f)` and `(Wy=0)`$\xrightarrow{rf}$`(f)`. Each of the two conclusions contradicts the assumption that initialization writes are coherence `co` ordered before non-initialization writes. Therefore this execution is forbidden by the JAM19 model. In the JAM21 model, we relax the `volint` order in `P1` and `P2` to include the situation of which the compiler inserts the `fullFence()` before Volatile accesses. Thus, under the new JAM21 model, this execution is allowed.

In summary, the JAM21 model has two main differences comparing to the JAM19 model. First, under the JAM21 model, when all memory accesses use Volatile mode, the execution is guaranteed to be sequentially consistent, whereas the old JAM19 model has the inconsistency issue we pointed out earlier. Second, when mixing Volatile and other access modes in a program, the new JAM21 model accommodates both the "leading fence" convention and the "trailing fence" convention so that the compiler is free to choose either one to implement.

Name	Power [SSA11]
volatile-non-sc.4.ppc	Sometimes
volatile-non-sc.5.ppc	Sometimes

Figure K.7: volatile-non-sc on Power with the incorrect compilation scheme

K.3 Compilation to Power

We translated the volatile-non-sc.4 and the volatile-non-sc.5 example to Power instructions according to the original compilation scheme:

The source code of the litmus tests in Power instructions can be found in Appendix K.4. Fig. K.7 shows the results of running the litmus tests with Power instructions on Herd7 using Power's memory model. Both of the executions are allowed under Power's memory model, which confirms the problem we addressed in this paper. The executions becomes forbidden if we change the `lwsync` instruction in the program to `hwsync`.

K.4 Source Code of litmus tests

In this section we provide the source code of the two examples that demonstrate the inconsistency issue we addressed in the paper. In addition, we include the same tests translated to Power instructions.

volatile-non-sc.4.litmus

```
Java volatile-non-sc.4
{
x = 0; y = 0;
0:X=x; 0:Y=y;1:X=x; 1:Y=y;
2:X=x; 2:Y=y;3:X=x; 3:Y=y;
}

Thread0 {
    Y.setVolatile(2);
    int r0 = X.getVolatile();
  }

Thread1 {
    X.setVolatile(1);
  }

Thread2 {
    int r0 = X.getVolatile();
    Y.setVolatile(1);
  }

Thread3 {
    int r0 = Y.getVolatile();
    int r1 = Y.getVolatile();
  }

exists
(0:r0=0 /\ 2:r0=1 /\ 3:r0=1 /\ 3:r1=2)
```

volatile-non-sc.5.litmus

```
Java volatile-non-sc.5

{

x = 0;y = 0;z = 0;

0:X=x;0:Y=y;0:Z=z;1:X=x;1:Y=y;1:Z=z;2:X=x;2:Y=y;2:Z=z;

3:X=x;3:Y=y;3:Z=z;4:X=x;4:Y=y;4:Z=z;

}


Thread0 {

X.setVolatile(1);

int r1 = Y.getVolatile();

}


Thread1 {

Y.setVolatile(1);

}


Thread2 {

int r1 = Y.getVolatile();
Z.setVolatile(1);
}

Thread3 {
Z.setVolatile(2);
int r1 = X.getVolatile();
}

Thread4 {
int r1 = Z.getVolatile();
int r2 = Z.getVolatile();
}

exists
(0:r1 = 0 /\ 2:r1 = 1 /\ 3:r1 = 0 /\ 4:r1 = 1 /\ 4:r2 = 2)
```

volatile-non-sc.4.ppc.litmus

```
PPC volatile-non-sc.4.ppc
{
0:r1=x; 0:r2=y;1:r2=y;
2:r1=x; 2:r2=y;3:r1=x;
}

P0           | P1           | P2           | P3           ;
li r3,2      | li r3,1      | li r3,1      | sync         ;
lwsync       | lwsync       | sync         | lwz r3,0(r1) ;
stw r3,0(r1) | stw r3,0(r2) | lwz r4,0(r2) | lwsync       ;
sync         | sync         | sync         | sync         ;
lwz r4,0(r2) |              | stw r3,0(r1) | lwz r4,0(r1) ;
lwsync       |              | sync         | lwsync       ;

exists
```

```
(0:r4=0 /\ 2:r4=1 /\ 3:r3=1 /\ 3:r4=2)
```

volatile-non-sc.5.ppc.litmus

```
PPC volatile-non-sc.5.ppc
{
0:r1=x; 0:r2=y;1:r2=y;
2:r2=y; 2:r3=z;3:r1=x; 3:r3=z;4:r3=z;
}
P0           | P1           | P2            | P3            | P4           ;
li r4,1      | li r4,1      | li r4,1       | li r4,2       | sync         ;
lwsync       | lwsync       | sync          | lwsync        | lwz r4, 0(r3);
stw r4,0(r1) | stw r4,0(r2) | lwz r5, 0(r2) | stw r4, 0(r3) | lwsync       ;
sync         | sync         | lwsync        | sync          | sync         ;
lwz r5,0(r2) |              | stw r4,0(r3)  | lwz r5, 0(r1) | lwz r5, 0(r3);
lwsync       |              | sync          | lwsync        | lwsync       ;

exists
(0:r5 = 0 /\ 2:r5 = 1 /\ 3:r5 = 0 /\ 4:r4 = 1 /\ 4:r5 = 2)
```

K.5 Full Trace and Litmus Test the example in Section. 3.1

The litmus test of the example of Fig. 3.2 is shown below. We labeled each memory instruction (in blue) in the litmus test for better readability of the trace. We obtained the trace by running the ppcmem tool by [SSA11] in the online interactive mode.

```
PPC volatile-non-sc.4.ppc
{
0:r1=x; 0:r2=y;
1:r2=y;
2:r1=x; 2:r2=y;
3:r1=x;
}

P0                | P1                | P2                | P3                 ;
li r3,2           | li r3,1           | li r3,1           | r:   sync          ;
a: lwsync         | f: lwsync         | m: sync           | s:   lwz r3,0(r1)  ;
b: stw r3,0(r1)   | g: stw r3,0(r2)   | n: lwz r4,0(r2)   | t:   sync          ;
c: sync           | h: sync           | o: lwsync         | t16: lwz r4,0(r1)  ;
d: lwz r4,0(r2)   |                   | p: stw r3,0(r1)   | t17: sync          ;
e: sync           |                   | q: sync           |                    ;

exists
(0:r4=0 /\ 2:r4=1 /\ 3:r3=1 /\ 3:r4=2)
```

One of the traces to show that this execution is allowed

```
(0:0) Commit reg or branch: li r3,2
(1:6) Commit reg or branch: li r3,1
(2:10) Commit reg or branch: li r3,1
(0:1) Commit barrier: lwsync: a:lwsync
(1:) Barrier propagate to thread: a:lwsync  to Thread 1
(2:) Barrier propagate to thread: a:lwsync  to Thread 2
(3:) Barrier propagate to thread: a:lwsync  to Thread 3
(1:7) Commit barrier: sync: f:Sync
(0:) Barrier propagate to thread: f:Sync  to Thread 0
(2:) Barrier propagate to thread: f:Sync  to Thread 2
(3:) Barrier propagate to thread: f:Sync  to Thread 3
Acknowledge sync: Sync f:Sync
(2:11) Commit barrier: sync: m:Sync
(0:) Barrier propagate to thread: m:Sync  to Thread 0
(1:) Barrier propagate to thread: m:Sync  to Thread 1
(3:) Barrier propagate to thread: m:Sync  to Thread 3
Acknowledge sync: Sync m:Sync
(3:16) Commit barrier: sync: r:Sync
```

```
(0:) Barrier propagate to thread: r:Sync  to Thread 0
(1:) Barrier propagate to thread: r:Sync  to Thread 1
(2:) Barrier propagate to thread: r:Sync  to Thread 2
Acknowledge sync: Sync r:Sync

(1:8) Commit write: stw r3,0(r2): g:W y=1 i:W x=0,j:W y=0
Write reaching coherence point: g:W y=1
(2:) Write propagate to thread: g:W y=1 to Thread 2
(2:12) Read from storage subsystem: lwz r4,0(r2) (from g:W y=1)
(2:12) Commit read: lwz r4,0(r2): n:R y=1
(2:13) Commit barrier: lwsync: o:Lwsync
(2:14) Commit write: stw r3,0(r1): p:W x=1 g:W y=1,i:W x=0
Write reaching coherence point: p:W x=1
(3:) Write propagate to thread: g:W y=1 to Thread 3
(3:) Barrier propagate to thread: o:Lwsync  to Thread 3
(3:) Write propagate to thread: p:W x=1 to Thread 3
(3:17) Read from storage subsystem: lwz r3,0(r1) (from p:W x=1)
(3:17) Commit read: lwz r3,0(r1): s:R x=1
(0:2) Commit write: stw r3,0(r1): b:W x=2 i:W x=0,j:W y=0
Write reaching coherence point: b:W x=2
(3:) Write propagate to thread: b:W x=2 to Thread 3
(0:3) Commit barrier: sync: c:Sync
(3:) Barrier propagate to thread: c:Sync  to Thread 3
(1:) Barrier propagate to thread: o:Lwsync  to Thread 1
(1:) Write propagate to thread: p:W x=1 to Thread 1
(1:) Write propagate to thread: b:W x=2 to Thread 1
(1:) Barrier propagate to thread: c:Sync  to Thread 1
(2:) Write propagate to thread: b:W x=2 to Thread 2
(2:) Barrier propagate to thread: c:Sync  to Thread 2
Acknowledge sync: Sync c:Sync
(0:4) Read from storage subsystem: lwz r4,0(r2) (from j:W y=0)
(0:4) Commit read: lwz r4,0(r2): d:R y=0
(0:) Write propagate to thread: g:W y=1 to Thread 0
```

```
(0:) Barrier propagate to thread: o:Lwsync  to Thread 0

(3:18) Commit barrier: sync: t:Sync

(0:) Barrier propagate to thread: t:Sync  to Thread 0

(1:) Barrier propagate to thread: t:Sync  to Thread 1

(2:) Barrier propagate to thread: t:Sync  to Thread 2

Acknowledge sync: Sync t:Sync

(3:19) Read from storage subsystem: lwz r4,0(r1) (from b:W x=2)

(3:19) Commit read: lwz r4,0(r1): t16:R x=2


(0:5) Commit barrier: sync: e:Sync

(1:) Barrier propagate to thread: e:Sync  to Thread 1

(2:) Barrier propagate to thread: e:Sync  to Thread 2

(3:) Barrier propagate to thread: e:Sync  to Thread 3

Acknowledge sync: Sync e:Sync

(1:9) Commit barrier: sync: h:Sync

(0:) Barrier propagate to thread: h:Sync  to Thread 0

(2:) Barrier propagate to thread: h:Sync  to Thread 2

(3:) Barrier propagate to thread: h:Sync  to Thread 3

Acknowledge sync: Sync h:Sync

(2:15) Commit barrier: sync: q:Sync

(0:) Barrier propagate to thread: q:Sync  to Thread 0

(1:) Barrier propagate to thread: q:Sync  to Thread 1

(3:) Barrier propagate to thread: q:Sync  to Thread 3

Acknowledge sync: Sync q:Sync

(3:20) Commit barrier: sync: t17:Sync

(0:) Barrier propagate to thread: t17:Sync  to Thread 0

(1:) Barrier propagate to thread: t17:Sync  to Thread 1

(2:) Barrier propagate to thread: t17:Sync  to Thread 2

Acknowledge sync: Sync t17:Sync


Result:

0:r4=0; 2:r4=1; 3:r3=1; 3:r4=2;
```

APPENDIX L

Relationship with Existing Soundness Definitions

In this section, we provide the formal definitions of the existing soundness definitions reviewed in §4.2 and show their relationships with our soundness definition. Since all of the definitions from existing works are based on sequential traces, we lift the definitions to execution graphs by treating each well-formed trace as a well-formed execution graph equipped with a `trace` order, which is a total order among all events in the execution graph. Note that we are focusing on the validity of each execution itself here, without considering its relationship with the reported bug. Therefore, we drop the requirement of composability.

Feasible Closure Feasible closure is defined in [HMR14] over sequential traces. We lift the definition to sequentially consistent execution graphs and use the same composition operation from Definition 14.

▶ Definition 29 (Feasible Closure)
Given a well-formed execution graph G_σ that is sequentially consistent with a linear order G_σ.`trace` over all events, the Feasible Closure of G_σ, feasible(G_σ), is the smallest set of well-formed execution graphs that includes G_σ and is closed under the following operations:

- Prefixes. If G_1 is an execution graph such that G_1.Evts $\subseteq G_2$.Evts is closed with respect to G_2.`trace` for some execution $G_2 \in$ feasible(G_σ), then $G_1 \in$ feasible(G_σ)
- Local Determinism. Assume that $G_1 \circ e_1, G_2 \in$ feasible(G_σ), $e_1.tid = t$, and $G_1|_t \approx G_2|_t$. Then
    - Branch. If $e_1.typ =$ br and $G_1|_t \cap G_1$.Rds $= G_2|_t \cap G_2$.Rds, then $G_2 \circ e_1 \in$ feasible(τ).
    - Read. If $e_1.typ = r$ and e_2 is a read event such that $e_2 \approx e_1$, then $G_2 \circ e_2 \in$ feasible(G_σ).
    - Write. If $e_1.typ = w$ then $G_2 \circ e_2 \in$ feasible(G_σ) with some write event e_2 such that

$e_2 \approx e_1$. If $G_1|_t \cap G_1.\mathsf{Rds} = G_2|_t \cap G_2.\mathsf{Rds}$, then $e_2.val = e_1.val$. Otherwise, $e_2.val = \hat{s}$ for some $\hat{s} \in \mathsf{Sym}$.

- Otherwise, $G_2 \circ e_1 \in \mathsf{feasible}(G_\sigma)$.

▶ Proposition 8
Let G_σ be a sequentially consistent input execution. Then for any $G_\rho \in \mathsf{feasible}(G_\sigma)$, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 5 with $C = W \cup R$ where

$$\begin{aligned} R &= \{r \in G_\sigma.\mathsf{Rds} \mid (\langle r, br\rangle \in G_\sigma.\mathsf{po} \text{ for some branch event } br \in G_\rho.\mathsf{Brs}) \\ &\quad \vee (\exists w \in C, \langle r, w\rangle \in G_\sigma.\mathsf{po})\} \\ W &= \{w \in G_\sigma.\mathsf{Wrts} \mid \exists r \in C, \langle w, r\rangle \in G_\sigma.\mathtt{rf}\} \end{aligned}$$

Notice that this definition of C satisfies $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ for event $e \in G_\rho$, because $\mathtt{ctrl} \subseteq \mathsf{po}; [\mathsf{Brs}]; \mathsf{po}$ and $\mathtt{data} \subseteq [\mathsf{Rds}]; \mathsf{po}; [\mathsf{Wrts}]$. By Proposition 2, the the composition operation $(\circ)$ preserves memory consistency. Since G_σ is sequentially consistent, we can inductively show that G_ρ is also sequentially consistent.

Relaxed CR We use the definition from [MKV18] for Relaxed CR, again lifted to execution graphs.

▶ Definition 30 (Relaxed CR)
Given a well-formed execution graph G_σ that is sequentially consistent, a well-formed execution G_ρ is a Relaxed CR of G_σ, if:

- G_ρ is sequentially consistent
- for each event $e \in G_\rho.\mathsf{Evts}$, if $\langle e', e\rangle \in G_\sigma.\mathsf{po}$, then $e' \in G_\rho.\mathsf{Evts}$ and $G_\rho.\mathsf{po} \subseteq G_\sigma.\mathsf{po}$

- for each read event $r \in G_\rho$.Evts, if r is not the last event in its thread, and $\langle w, r\rangle \in G_\sigma.\texttt{rf}$, then $w \in G_\rho$.Evts and $\langle w, r\rangle \in G_\rho.\texttt{rf}$ for each pair of such read and write events.

▶ Proposition 9
Let G_σ be a sequentially consistent input execution. Then for any Relaxed CR G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 5 with $C = W \cup R$ where

$$R = \{r \in G_\sigma.\mathsf{Rds} \mid \langle r, e\rangle \in G_\rho.\mathsf{po} \text{ for some non-branch event } e \in G_\rho.\mathsf{Evts}\ \}$$
$$W = G_\sigma.\mathsf{Wrts}$$

Notice that this definition of C satisfies $C_\sigma \subseteq C$ for event $e \in G_\rho$, since all of the rules (IV, V, VI in Definition 19) that add a read event r into C_σ require some non-branch event e' to be po-ordered after r in G_ρ. Lastly, G_ρ is sequentially consistent by definition.

It's easy to see that the concrete set C of Feasible Closure is a subset of the concrete set C of Relaxed CR, while both require the execution to be well-formed and sequentially consistent. Hence, any Relaxed CR is also in the Feasible Closure of the input execution.

Correct Reordering Correct Reordering is defined as the following.

▶ Definition 31 (Correct Reordering)
Given a well-formed execution G_σ that is sequentially consistent, a well-formed execution G_ρ is a Correct Reordering of G_σ if:

- G_ρ is sequentially consistent,
- for each event $e \in G_\rho$.Evts, if $\langle e', e\rangle \in G_\sigma.(\mathsf{po} \cup \texttt{rf})$, then $e' \in G_\rho$.Evts,
- $G_\rho.\texttt{rf} = G_\sigma.\texttt{rf} \cap (G_\rho.\mathsf{Wrts} \times G_\rho.\mathsf{Rds})$,
- $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po} \cap (G_\rho.\mathsf{Evts} \times G_\rho.\mathsf{Evts})$.

▶ Proposition 10
Let G_σ be a sequentially consistent input execution. Then for any Correct Reordering G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 5 with $C = G_\sigma.\text{Wrts} \cup G_\sigma.\text{Rds}$. It is obvious that this definition of C satisfies $C_\sigma \subseteq C$ for each event $e \in G_\rho$. Finally, G_ρ is sequentially consistent by definition.

Sync-Preserving CR The formal definition of Sync-Preserving CR is given by

▶ Definition 32
Given a well-formed execution G_σ that is sequentially consistent, a well-formed execution G_ρ is a Sync-Preserving CR of G_σ if:

- G_ρ is a Correct Reordering of G_σ
- $G_\rho.\texttt{sync} \subseteq G_\sigma.\texttt{sync}$

By definition, any Sync-Preserving CR is a Correct Reordering of the input execution, hence the following proposition.

▶ Proposition 11
Let G_σ be a sequentially consistent input execution. Then for any Sync-Preserving CR G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

Interestingly, if we want to show a Sync-Preserving CR is executable without considering the obvious relation with Correct Reordering, the concrete set C for Sync-Preserving CR is the same as Correct Reordering. This is because Sync-Preserving CR only differ from Correct Reordering in the memory orders among the events, whereas executability is *not* affected by memory orders other than po.

APPENDIX M

Language Semantics for Chapter 4

In this section we provide the operational semantics for our language. The semantics are intended to be straightforward and unsurprising.

Recall that a thread state $st \in$ State for a thread t is a tuple of the form

$$st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^{\theta} \rangle$$

Helper Functions We start by defining the helper functions.

Given a map $\theta : G.\mathsf{Rds} \to \mathsf{Sym}$, we define the function $val^{\theta} : \wp(G.\mathsf{Rds}) \to (\mathsf{Sym} \to \mathsf{Val})$ as the following.

$$\begin{aligned} val^{\theta}[] &\triangleq \lambda s.0 \\ val^{\theta}(r :: tl) &\triangleq \lambda s.\ \texttt{if}\ s = \theta(r)\ \texttt{then}\ r.val\ \texttt{else}\ val^{\theta}(tl)(s) \end{aligned}$$

The substitution function $subst : \mathsf{SymExpr} \to (\mathsf{Sym} \to \mathsf{Val}) \to \mathsf{Expr}$ replaces symbols with concrete values in an expression.

In addition, the function $mkExpr : \mathsf{Expr} \to (\mathsf{Reg} \to \mathsf{SymExpr}) \to \mathsf{SymExpr}$ converts a concrete expression to a symbolic expression using a given map $\Phi^{\theta} : \mathsf{Reg} \to \mathsf{SymExpr}$.

The function $newSym : \wp(\mathsf{Sym}) \to \mathsf{Sym}$ returns a fresh new symbol that does not occur in the given set of existing symbols.

Finally, we use a the same function add_G defined from [PLV19] but without read-modify-write operations and address dependencies (we assume all memory locations are fixed).

Semantic Rules Recall that we are given a map $loadVal : Load \rightarrow \mathsf{Val}$ for each load instruction of the form $r := [x]$ with $r \in \mathsf{Reg}$ and $x \in \mathsf{Loc}$. In this section, we use the helper functions defined previously and the $loadVal$ map to define the semantic rules for our language.

We use the notation $st_1 \xrightarrow{instr}_t st_2$ to represent a state transition in thread t from st_1 to st_2 where $st_1.sprog(st_1.pc) = instr \in \mathsf{Instr}$ and $st_1.sprog = st_2.sprog$.

$$\frac{\begin{array}{c} st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi[r \mapsto st_1.\Phi(e)] \\ st_2.G = st_1.G \\ st_2.\Psi = st_1.\Psi[r \mapsto st_1.\Psi(e)] \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta[r \mapsto mkExpr(e, st_1.\Phi^\theta)] \end{array}}{st_1 \xrightarrow{r:=e}_t st_2} \quad (\textsc{Assignment})$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\mathsf{Evts}|, \mathsf{Rds}, loadVal(i), x\rangle \\ \hat{s} = newSym(codomain(st_1.\theta)) \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi[r \mapsto loadVal(i)] \\ st_2.G = add(st_1.G, evt, \varnothing, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi[r \mapsto \{evt\}] \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta[evt \mapsto \hat{s}] \\ st_2.\Phi^\theta = st_1.\Phi^\theta[r \mapsto \hat{s}] \end{array}}{st_1 \xrightarrow{r:=[x]}_t st_2} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\mathsf{Evts}|, \mathsf{Wrts}, st_1.\Phi(e), x\rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \Psi(e), st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{[x]:=e}_t st_2} \text{(Store)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\mathsf{Evts}|, \mathsf{Acqs}, -, l\rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \varnothing, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\mathtt{lock}(l)}_t st_2} \text{(Lock)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\mathsf{Evts}|, \mathtt{Br}, -, -\rangle \\ st_1.\Phi(e) = 0 \Rightarrow st_2.pc = st_1.pc + 1 \\ st_1.\Phi(e) \neq 0 \Rightarrow st_2.pc = n \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \varnothing, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \cup st_1.\Psi(e) \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\mathtt{if}\ e\ \mathtt{goto}\ n}_t st_2} \text{(If-Goto)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\mathsf{Evts}|, \mathsf{Rels}, -, l\rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \varnothing, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\mathtt{unlock}(l)}_t st_2} \text{(Unlock)}$$

APPENDIX N

Proofs for Soundness Properties in §4.4

In this section, we present the proofs of §4.4.

▶ Proposition 1
Let G_ρ be a well-formed execution graph such that $G_\rho \in [\![P]\!]$, and b be a bug sequence. If $G_\rho \rhd b$, then $(G_\rho \circ b) \in [\![P]\!]$.

Proof. Note that $(G_\rho \circ b)$ is a symbolic execution graph since all events in b have symbolic values. From $G_\rho \in [\![P]\!]$, we know that for each thread $t \in G_\rho.\mathsf{Thrd}$, there is a state transition path $st'_0 \to \cdots \to st'_m$ generating $G_\rho|_t$ where $st_0 = st'_0$ and $st_i \approx st'_i$ for $st_i \in Path(G_\sigma|_t)$. Let e_1 be the first event and e_n be the last event in $b|_t$. Then there is a set $\{\delta_b(e_1), \ldots, \delta_b(e_n)\} \subseteq G_\sigma|t.\mathsf{Evts}$. Let $st_j \to \cdots \to st_n$ be the subsequence of transition path that generates $\{\delta_b(e_1), \ldots, \delta_b(e_n)\}$. We construct a set of states $\{st'_j, \ldots, st'_n\}$ as follows. For each $i \in j \ldots n$, we first set $st'_i.sprog = st_i.sprog$, $st'_i.pc = st_i.pc$, and $st'_i.\Phi^\theta = st_i.\Phi^\theta$. For each read event $r' \in b|_t.\mathsf{Rds}$, there exists a read event $r = \delta_b(r')$ in $G_\sigma|_t.\mathsf{Rds}$ such that $r' \approx r$. Set $st'_i.\theta$ be a symbol map such that $st'_i.\theta(r') = st_i.\theta(r)$ for each read event r'. Then $st'_i.\Psi$ and $st'_i.ctrl$ are derived from $st_i.\Psi$ and $st_i.ctrl$ respectively by replacing each r with r'. Finally, $st'_i.G$ is derived from $st_i.G$ by replacing every event e with e', where $e = \delta_b(e')$. By this construction, since b is sequentially consistent, the set $\{st'_j, \ldots, st'_n\}$ is $sqsubseteq$-ordered. In addition, from the No Skipping condition, we know that there is $e_q \in G_\rho$ such that $\langle \delta_\rho(e_q), \delta_b(e_1)\rangle \in G_\sigma.\mathsf{po}$. Hence, the set $\{st'_0, \ldots, st'_m, st'_j, \ldots, st'_n\}$ is $sqsubseteq$-ordered as well. For state st'_{m-1}, if $st'_{m-1}.sprog(st'_{m-1}.pc) = \texttt{if}\ expr\ \texttt{goto}\ n$, for each $r \in st_{m-1}.\Psi(expr)$, we have $\langle \delta_\rho(r), \delta_b(e_k)\rangle \in G_\sigma.\texttt{ctrl}$ for all $k \in 1 \ldots n$. From the Same Control Flow condition, we know that $\delta_\rho(r) = r$. Hence, $st_{m-1}.\Phi(expr) = st_{m-1}.\Phi(expr)$.

From the assumption that $G_\rho \in [\![P]\!]$ and that there is no concrete event in b, we can infer that $st'_0 \rightarrow \cdots \rightarrow st'_m \rightarrow st'_j \rightarrow \cdots \rightarrow st'_n$ is a valid transition path generating $(G_\rho \circ b)$. Thus, $(G_\rho \circ b) \in [\![P]\!]$. ☐

▶ Proposition 2
Let G_ρ be a well-formed execution graph such that G_ρ is $\mathcal{M}$-consistent, and b be a sequence of events. Then $(G_\rho \circ b)$ is $\mathcal{M}$-consistent.

Proof. Towards contradiction, suppose there is a $\mathtt{ppo} \cup \mathtt{com} \cup (\mathtt{po}^?; [L]; \mathtt{po}^?) \cup \mathtt{sync}$ cycle in $(G_\rho \circ b)$. Since G_ρ is $\mathcal{M}$-consistent and b is a sequentially consistent, there must be an ordering edge from an event of b to an event of G_ρ. Since $(G_\rho \circ b).\mathtt{po}$ orders all events of G_ρ before events of b on the same threads, this back edge is not a $\mathtt{po}$ edge. By the definition of composition, we can further rule out $\mathtt{sync}$, $\mathtt{co}$, and $\mathtt{rf}$ edge for similar reasons. If it is an $\mathtt{fr}$ edge, then it means there is a write event $w \in b.\mathsf{Wrts}$ such that $\langle w, r\rangle \in b.\mathtt{rf}$ for some read event $r \in b.\mathsf{Rds}$ and $\langle w, w'\rangle \in (G_\rho \circ b).\mathtt{co}$. However, it contradicts with the definition of $(G_\rho \circ b).\mathtt{co}$. Thus, $(G_\rho \circ b)$ is $\mathcal{M}$-consistent. ☐

Lemma 3 is proved using two helper lemmas, Lemma 18 and Lemma 19. We begin by providing their proofs.

▶ Lemma 18
Let $st_1, st_2, st'_1, st'_2 \in \mathsf{State}$ be four valid states and $st_1.sprog(st_1.pc) \neq$ `if` e `goto` n. If $st_1 \approx st'_1$, $st_2 \approx st'_2$ via the same map δ and $st_1 \rightarrow st_2$, then $st'_1 \rightarrow st'_2$.

Proof. Let $st_1 = \langle sprog_1, pc_1, \Phi_1, G_1, \Psi_1, ctrl_1, \theta_1, \Phi^\theta_1\rangle$, $st_2 = \langle sprog_2, pc_2, \Phi_2, G_2, \Psi_2, ctrl_2, \theta_2, \Phi^\theta_2\rangle$, $st'_1 = \langle sprog'_1, pc'_1, \Phi'_1, G'_1, \Psi'_1, ctrl'_1, \theta'_1, \Phi^{\theta\prime}_1\rangle$, and $st'_2 = \langle sprog'_2, pc'_2, \Phi'_2, G'_2, \Psi'_2, ctrl'_2, \theta'_2, \Phi^{\theta\prime}_2\rangle$.

Recall that each valid state satisfies the invariant $\forall r \in \mathsf{Reg},\ subst(\Phi^\theta(r), val^\theta(\Psi(r))) = \Phi(r)$.

It's obvious that $sprog_1 = sprog_2 = sprog'_1 = sprog'_2$.

Due to the data-abstract equivalence relations and the assumption that $st_1.sprog(st_1.pc) \neq$ `if` e `goto` n, we have $pc_1 = pc'_1$ and $pc_2 = pc'_2$. Because $sprog_1(pc_1) \neq$ `if` e `goto` n, we have $pc_2 = pc_1 + 1$. Hence, $pc'_2 = pc'_1 + 1$.

For the execution graph, we have $G_2 = add(G_1, e)$ for some event e. Therefore, for each $e' \in G_2$ that $e' \neq e$, we know that $e' \in G_1$, which implies that $\delta(e') \in G_1$. For e, we have $e \notin G_1$ which implies that $\delta(e) \notin G'_1$. In addition, because $G_2 \approx G'_2$, we know that $\delta(e) \in G'_2$. Therefore, we can infer that $G'_2 = add(G'_1, \delta(e))$.

For the $ctrl$ set, because $sprog(pc_1) \neq$ `if` e `goto` n, we have $ctrl_1 = ctrl_2$. Because $st_2 \approx st'_2$ for each read event $e \in ctrl_2$, we have $\delta(e) \in ctrl'_2$. Similarly, for each read event $e \in ctrl'_1$, we have $\delta^{-1}(e) \in ctrl_1$. Hence, for each read event $e \in ctrl'_1, \delta(\delta^{-1}(e)) = e \in ctrl'_2$. Similarly, for each read event $e \in ctrl'_2, \delta(\delta^{-1}(e)) = e \in ctrl'_1$. Therefore, $ctrl'_1 = ctrl'_2$.

Let $i = sprog_1(pc_1)$. We do a case analysis on i to establish the relations on Φ, Φ^θ and Ψ.

- $i =$ `lock(`l`)` or $i =$ `unlock(`l`)`. Then we have $\Phi_2 = \Phi_1$ and $\Psi_1 = \Psi_2$. It's obvious that $\Phi^\theta_1 = \Phi^{\theta\prime}_1 = \Phi^\theta_2 = \Phi^{\theta\prime}_2$. Because the data-abstract equivalence is established via the same function δ, for each register $r \in$ Reg, we have $\Phi'_2(r) = subst(\Phi^{\theta\prime}_1(r), val^\theta(\Psi'_1(r))) = \Phi'_1(r)$. Hence, $\Phi'_1 = \Phi'_2$. Given that $\Psi_1 = \Psi_2$, for each register $r \in$ Reg, $\Psi_1(r) = \Psi_2(r)$. Lifting the function δ, we have $\delta(\Psi_1(r)) = \delta(\Psi_2(r))$ for each $r \in$ Reg. Therefore, $\Psi'_1 = \Psi'_2$.

- $i =$ r `:=` e. Then we have $\Phi_2 = \Phi_1[r \mapsto \Phi_1(e)]$ and $\Phi^\theta_2 = \Phi^\theta_1[r \mapsto \hat{e}]$ where $\hat{e} = mkExpr(e, \Phi^\theta_1)$. In addition, we have $\Psi_2 = \Psi_1[r \mapsto \Psi_1(e)]$. From $st_2 \approx st'_2$, we have $\Phi^\theta_2 = \Phi^{\theta\prime}_2$. From $st_1 \approx st'_1$, we have $\Phi^\theta_1 = \Phi^{\theta\prime}_1$. Since for each read event $r \in G_1$.Rds, $\theta_1(r) = \theta'_1(\delta(r))$, we have $\Phi^{\theta\prime}_2 = \Phi^{\theta\prime}_1[r \mapsto \hat{e}]$. For each $r' \neq r$, it's obvious that $\Phi'_2(r') = \Phi'_1(r')$. We know that r does not occur on the right-hand-side of the assignment, therefore, $\Psi'_1(e) = \Psi'_2(e)$ and hence $\Phi'_2(r) = subst(\Phi^{\theta\prime}_2(r), val^\theta(\Psi'_1(e))) = subst(\Phi^{\theta\prime}_1[r \mapsto \hat{e}](r), val^\theta(\Psi'_1(e))) = subst(\hat{e}, val^\theta(\Psi'_1(e))) = \Phi'_1(e)$. Therefore, $\Phi'_2 = \Phi'_1[r \mapsto \Phi'_1(e)]$. Given that $\Psi_2 = \Psi_1[r \mapsto \Psi_1(e)]$, for registers $r' \neq r$, $\Psi'_2(r') = \delta(\Psi_2(r')) = \delta(\Psi_1(r')) = \Psi'_1(r')$. For the register r, we have $\Psi_2(r) = \Psi_1(e)$. Therefore, $\Psi'_2(r) = \delta(\Psi_2(r)) = \delta(\Psi_1(e)) = \Psi'_1(e)$. Thus,

$\Psi'_2 = \Psi'_1[r \mapsto \Psi'_1(e)]$.

- $i =$ `r := [x]`. Then we have $\Phi_2 = \Phi_1[r \mapsto v]$ for some concrete value v and $\Phi^\theta_2 = \Phi^\theta_1[r \mapsto \hat{v}]$ for some symbolic value $\hat{v}$. From $st_2 \approx st'_2$ and $st_1 \approx st'_1$, we have $\Phi^{\theta\prime}_2 = \Phi^{\theta\prime}_1[r \mapsto \hat{v}]$. Now we want to show that there exists a concrete value v' such that $\Phi'_2 = \Phi'_1[r \mapsto v']$. Note that at this stage of generating execution graph, we do not restrict the value of each event to be consistent with the memory model or well-formed. Therefore, any value from the domain Val is eligible to be given to a read event. Since the domain Val is not empty, we can be sure that there exists a value $v' \in$ Val such that $\delta(e).val = v'$, where e is the read event generated from st_1 to st_2 with $G_2 = add(G_1, e)$. Because we have already established that $\Phi^{\theta\prime}_2 = \Phi^{\theta\prime}_1[r \mapsto \hat{v}]$ and $\theta_2 \approx \theta'_2$, we have $\Phi'_2 = \Phi'_1[r \mapsto v']$ for some $v' \in$ Val. From the data-abstract equivalence relation among states, we can easily see that $\Psi'_2 = \Psi'_1[r \mapsto \{\delta(e)\}]$ via a similar set of reasonings for other cases.
- $i =$ `[x] := e`. Then we have $\Phi_1 = \Phi_2$, $\Phi^\theta_1 = \Phi^\theta_2$ and $\Psi_1 = \Psi_2$. We immediately have $\Phi^\theta_1 = \Phi^{\theta\prime}_1 = \Phi^\theta_2 = \Phi^{\theta\prime}_2$. In addition, from the data-abstract equivalence relations, we get that $\Psi'_2 = \Psi'_1$. Because the data-abstract equivalence relations are established via the same function δ, for each register $r \in$ Reg, $val^\theta(\Psi'_2(r)) = val^\theta(\Psi'_1(r))$. Therefore, $\Phi'_2(r) = subst(\Phi^{\theta\prime}_2(r), val^\theta(\Psi'_1(r))) = \Phi'_1(r)$.

With the relations on each of the components of st'_1 and st'_2 established, we can conclude that $st'_1 \to st'_2$. $\square$

As we have seen in the proof, we are able to establish the state transition relation with only the data-abstract equivalence relations among states when there is no if-goto instruction because the control flow of the program does not depend on the concrete values in those cases. On the other hand, there is one extra condition needed on states with if-goto instructions because this is where the control flow becomes dependent on the values. We have the following lemma.

▶ Lemma 19
Let $st_1, st_2, st'_1, st'_2 \in$ State be four valid states and $st_1.sprog(st_1.pc) =$ `if` e `goto` n. If $st_1 \approx st'_1$, $st_2 \approx st'_2$ via the same bijective function δ, $st_1 \to st_2$ and $\Phi_1(e) = \Phi'_1(e)$, then $st'_1 \to st'_2$.

Proof. If $\Phi_1(e) = \Phi'_1(e) = 0$, then $pc_2 = pc'_2 = pc_1 + 1 = pc'_1 + 1$. If $\Phi_1(e) = \Phi'_1(e) \neq 0$, then $pc_2 = pc'_2 = n$. Therefore, we can establish that $\Phi'_1 = 0 \Rightarrow pc'_2 = pc'_1 + 1$ and $\Phi'_1 \neq 0 \Rightarrow pc'_2 = n$. We can establish the relations of other components of st'_1 and st'_2 using a similar proof as the one for Lemma 18. Thus, $st'_1 \to st'_2$. □

We are now ready to proof Lemma 3.

▶ Lemma 3
Let G_ρ, G_σ be well-formed execution graphs and $G_\sigma \in [\![P]\!]$. If for each thread $t \in G_\rho$.Thrd, $t \in G_\sigma$.Thrd and there is a $\sqsubseteq$-ordered set $\{st'_0, \ldots, st'_m\}$ such that for $i \in 0 \ldots m$,

- each state st'_i satisfies the invariant $subst(\Phi^\theta(r), val^\theta(\Psi(r))) = \Phi(r)$ for $r \in$ Reg,
- for each st'_i there exists a state $st_i \in Path(G_\sigma|_t)$ with $st_i \approx st'_i$ and $st_0 = st'_0$,
- for each st'_i if $st'_i.sprog(st'_i.pc) =$ `if` $expr$ `goto` k then $st_i.\Phi(expr) = st'_i.\Phi(expr)$,
- $st'_m.G = G_\rho|_t$,

then $G_\rho \in [\![P]\!]$.

Proof. Induction on the size of the set $\{st'_0, \ldots, st'_m\}$. For the base case, since $st'_0 = st_0$ is the initial state, it's obvious that the lemma holds. For the inductive step, assume that the lemma holds for $\{st'_0, \ldots, st'_i\}$. That is, given such set of states, we have $st'_0 \to^*_t st'_i$. We want to prove the same property holds for st'_{i+1}. There are two cases to analyze. If $st'_i.sprog(st'_i.pc) =$ `if` e `goto` n, then we can apply Lemma 19. Otherwise, we can apply Lemma 18. By the transitivity of state transition relation, we have $st'_0 \to^*_t st'_{i+1}$. Thus, we can conclude that $st'_0 \to^*_t st'_m$. Hence, $G_\rho \in [\![P]\!]$ by Definition 13. □

▶ Lemma 4
Let $\hat{G}_\rho$ be a well-formed symbolic execution graph and G_σ be a concrete input execution graph such that $G_\sigma \in [\![P]\!]$. If $\hat{G}_\rho$ satisfies the following conditions:

- there is a map $\delta : \hat{G}_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ such that for each event $e \in \hat{G}_\rho.\mathsf{Evts}$, $\delta(e) \approx e$ and if $e.val \in \mathsf{Val}$ (i.e., $e.val$ is concrete), then $\delta(e) = e$.
- if $\langle e_1, e_2 \rangle \in G_\sigma.\mathsf{po}$ and $e_2 = \delta(e'_2)$ for some $e'_2 \in \hat{G}_\rho.\mathsf{Evts}$, then there is an event $e'_1 \in \hat{G}_\rho.\mathsf{Evts}$ such that $e_1 = \delta(e'_1)$ and $\langle e'_1, e'_2 \rangle \in \hat{G}_\rho.\mathsf{po}$,
- for each thread $t \in \hat{G}_\rho.\mathsf{Thrd}$ and each event $e \in \hat{G}_\rho|_t.\mathsf{Evts}$, if there is a read event $r \in \hat{G}_\rho|_t.\mathsf{Rds}$, such that $\langle \delta(r), \delta(e) \rangle \in G_\sigma.\mathtt{ctrl}$, then $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete),
- for each write event $w \in \hat{G}_\rho.\mathsf{Wrts}$, if $w.val \in \mathsf{Val}$ (i.e., $w.val$ is concrete), then for all $r \in \hat{G}_\rho.\mathsf{Rds}$ such that $\langle \delta(r), \delta(w) \rangle \in G_\sigma.\mathtt{data}$, $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete).

then $\hat{G}_\rho \in [\![P]\!]$.

Proof. For each thread $t \in \hat{G}_\rho.\mathsf{Thrd}$, we know that $t \in G_\sigma.\mathsf{Thrd}$ because the map $\delta : \hat{G}_\rho.\mathsf{Evts} \to G_\sigma.\mathsf{Evts}$ maps each event $e \in G_\rho.\mathsf{Evts}$ to some event $\delta(e) \in G_\sigma.\mathsf{Evts}$ and $\delta(e) \approx e$, which implies $\delta(e).tid = e.tid$. Because $G_\sigma \in [\![P]\!]$, there is a valid transition path generating $G_\sigma|_t$. Let st_m be the state right after e_n is generated (note that $n \leq m$ because there may be internal transitions that do not emit any memory event). Then there is a set $\{st_0, \ldots, st_m\}$ that is $\sqsubseteq$-ordered. The goal is to derive a set of states $\{st'_0, \ldots, st'_m\}$ from $\{st_0, \ldots, st_m\}$ such that the derived states satisfy the requirements in Lemma 3.

We start by setting $st'_0 = st_0$, where st_0 is the initial state. For each state st_i where $i \in 1 \ldots m$, there is a state st'_i constructed in the following way.

For each state st_i, we first set $st'_i.sprog = st_i.sprog$, $st'_i.pc = st_i.pc$, and $st'_i.\Phi^\theta = st_i.\Phi^\theta$. That is, the sequential program instructions, program counter, and the symbolic expressions recorded for each register on each state are directly copied from the original states. For each read event $r' \in G_\rho|_t.\mathsf{Rds}$, there exists a read event $r = \delta(r')$ in $G_\sigma|_t.\mathsf{Rds}$ such that $r' \approx r$. Set $st'_i.\theta$ be a symbol map such that $st'_i.\theta(r') = st_i.\theta(r)$ for each read event r'. If r' is a concrete event, we can then use $st'_i.\theta$ to compute the concrete values of registers by plugging in $r'.val$. Then $st'_i.\Psi$ and

$st'_i.ctrl$ are derived from $st_i.\Psi$ and $st_i.ctrl$ respectively by replacing each r with r'. Finally, $st'_i.G$ is derived from $st_i.G$ by replacing every event e with e', where $e = \delta(e')$. By this construction, for each state st_i and st'_i, we have $st_i \approx st'_i$ and the invariant $subst(\Phi^\theta(reg), val^\theta(\Psi(reg))) = \Phi(reg)$ for $reg \in \mathsf{Reg}$ holds if $\Phi(reg) \in \mathsf{Val}$. In addition, for each $\langle e_1, e_2\rangle \in \hat{G}_\rho.\mathsf{po}$, there is $\langle \delta(e_1), \delta(e_2)\rangle \in G_\sigma.\mathsf{po}$. Therefore, $\{st'_0, \ldots, st'_m\}$ is $\sqsubseteq$-ordered.

For each state st'_i such that $st'_i.sprog(st'_i.pc) = \texttt{if}\ expr\ \texttt{goto}\ n$, let e'_j be the event generated right before st'_i. Then by the language semantics, for each $r \in st'_i.\Psi(expr)$, we have $\langle \delta(r), \delta(e'_k)\rangle \in G_\sigma.\texttt{ctrl}$ for all $k > j$. From the assumption, we know that $r.val \in \mathsf{Val}$ and $\delta(r) = r$. Hence, $st'_i.\Phi(expr)$ is concrete and $st'_i.\Phi(expr) = st_i.\Phi(expr)$.

Lastly, for each state st'_i such that $st'_i.sprog(st'_i.pc) = [x]\ \texttt{:=}\ expr$, let w be the write event generated after st'_i. If $w.val \in \mathsf{Val}$, then by the semantics of store instruction, $st'_i.\Phi(expr) \in \mathsf{Val}$ has to be concrete. Hence, for all $r \in st'_i.\Psi(expr)$, $r.val \in \mathsf{Val}$ has to be concrete. Note that for all $r \in st'_i.\Psi(expr)$, there is $\langle \delta(r), \delta(w)\rangle \in G_\sigma.\texttt{data}$, and we have the assumption that $\delta(r) = r$. Therefore, $st'_i.\Phi(expr) = st_i.\Phi(expr) \in \mathsf{Val}$. $\square$

▶ Theorem 22 (DRF-SC)
Given a program P, if all its SC-consistent executions are data-race-free or only have volatile-races, then the set of all Jam21-consistent executions of P coincide with the set of SC-consistent executions.

Proof. Towards contradiction, suppose not. Then there is an execution of P that is not sequentially consistent but is M-consistent for some MCA memory model M. Hence, there is a $(\mathsf{po} \cup \texttt{com} \cup \texttt{sync})^+$ cycle in the execution where at least one of the edges is not $\texttt{ppo}$ edge. By the transitive nature of po order, for each event in the cycle is either a source or a target or both of a communication $\texttt{com}$ edge.

First, note that $\texttt{com}$ relates conflicting events. Hence, for each two events related by $\texttt{com}$, there is a lock protecting the two events. In other words, for each $\langle e_1, e_2\rangle \in \texttt{com}$, there are lock events $\texttt{acq}_1(\texttt{l})$ and $\texttt{rel}_1(\texttt{l})$ enclosing e_1 and $\texttt{acq}_2(\texttt{l})$ and $\texttt{rel}_2(\texttt{l})$ enclosing e_2. In addition, we can also infer that $\texttt{rel}_1(\texttt{l}) \xrightarrow{\texttt{sync}} \texttt{acq}_2(\texttt{l})$ since critical sections of the same lock are linearly ordered and the other direction would immediately yield a contradiction.

In addition, for each two events $e_1 \xrightarrow{\texttt{po}} e_2$ but not $e_1 \xrightarrow{\texttt{ppo}} e_2$, they are enclosed in the same one or more critical sections. If not, then either they are not in the same critical section of some lock or they are both not in any critical section. In the former case, there would be a lock event po ordered between e_1 and e_2. By our new definition for $\texttt{ppo}$, this would result in $e_1 \xrightarrow{\texttt{ppo}} e_2$, which is a contradiction. In the latter case, note that each event in this cycle is either a source or a target or both of a communication $\texttt{com}$ edge relating conflicting events. The hypothesis states that each conflicting events are protected by the same lock, which implies the existence of a critical section enclosing every event in the cycle.

Since for each two events $e_1 \xrightarrow{\texttt{po}} e_2$ but not $e_1 \xrightarrow{\texttt{ppo}} e_2$, they are enclosed in the same one or more critical sections, from the new definition of $\texttt{ppo}$, we can infer that $\texttt{acq} \xrightarrow{\texttt{ppo}} \texttt{rel}$ for *any* acquire event and release event (they may be lock events for different locks) enclosing e_1 and e_2.

Hence, if there is a $(\texttt{po} \cup \texttt{com} \cup \texttt{sync})^+$ cycle in the execution graph, there is also a $(\texttt{ppo} \cup \texttt{com} \cup \texttt{sync})^+$ cycle in the same graph, contradicting with the assumption that the execution graph is M-consistent. Thus, any sound execution of the same program must be sequentially consistent. □

APPENDIX O

Proofs of §4.5

▶ PROPOSITION 4
If $\langle S_\sigma, C_\sigma\rangle$ enables a bug sequence b, then $\hat{G}_\rho$ is well-formed up to concrete events and $\hat{G}_\rho \rhd b$.

Proof. Since G_σ.po is well-formed, it follows that $\hat{G}_\rho$.po is also well-formed. For each read event $r \in \hat{G}_\rho$.Rds, if $r.val \in$ Val, then it means $r = \delta(r) \in C_\sigma$. Since $\langle S_\sigma, C_\sigma\rangle$ enables the bug sequence b, by (VII), there exists a write event $w \in \hat{G}_\rho$.Wrts such that $\delta(w) = w \in C_\sigma$, $w.loc = r.loc$, and $w.val = r.val$. Hence, $\hat{G}_\rho$ is read feasible. In addition, (III) ensures $\hat{G}_\rho$ is lock feasible.

(VIII) ensures the premise for composability. No Skipping is satisfied by (I). Same Control Flow is satiesfied by (IV). Hence $\hat{G}_\rho \rhd b$. □

▶ PROPOSITION 5
If $\langle S_\sigma, C_\sigma\rangle$ enables a bug sequence b, and $G_\sigma \in [\![P]\!]$, then $\hat{G}_\rho \in [\![P]\!]$.

Proof. It's obvious that the map δ satisfies the requirement $\delta(e) \approx e$ for each $e \in \hat{G}_\rho$.Evts. For each event $e_1 \in \hat{G}_\rho$.Evts, let $e'_1 = \delta(e)$. Then $e'_1 \in S_\sigma$. By (II), if there is $\langle e'_2, e'_1\rangle \in G_\sigma$.po, then $e'_2 \in S_\sigma$. Hence, there is $e_2 \in \hat{G}_\rho$.Evts such that $\delta(e_2) = e'_2$ and $\langle e_2, e_1\rangle \in G_\rho$.po. For each thread $t \in \hat{G}_\rho$.Thrd, let e_t be the last event on thread t. By (V), if there is a read event $r \in G_\sigma$.Evts such that $r = \delta(r')$ for some $r' \in \hat{G}_\rho$.Rds and $\langle r, \delta(e_t)\rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$. Then $r = \delta(r') = r'$. A write event w has a concrete value if and only if $\delta(w)$ is included in C_σ. From (VI), for each $\delta(w) \in C_\sigma$, if there is $\langle \delta(r), \delta(w)\rangle \in G_\sigma.\texttt{data}$, then $\delta(r) \in C_\sigma$, which means $\delta(r) = r \in \hat{G}_\rho$.Rds. Therefore, by LEMMA 4, $\hat{G}_\rho \in [\![P]\!]$. □

▶ Lemma 5

Let $\hat{G}_\rho = (\mathsf{Evts}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evts}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{rf}|_C)$ where C is the set of concrete read and write events of $\hat{G}_\rho$ such that $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ for a bug sequence b, and $\delta(e) = e$ for each $e \in C$. Then $\hat{G}_\rho$ is read-feasible up to C and executable.

Proof. It is clear that $\hat{G}_\rho$ is read feasible by the fact that $\hat{G}_\rho$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{rf}|_C)$ for all concrete events $e \in C$. Because $\hat{G}_\rho$ is downward-closed with respect to $G_\sigma.\mathsf{po}$, for each $e_2 \in \hat{G}_\rho.\mathsf{Evts}$, if there is $\langle e_1', \delta(e_2)\rangle \in G_\sigma.\mathsf{po}$, then there is $e_1 \in \hat{G}_\rho.\mathsf{Evts}$ such that $e_1' = \delta(e_1)$ and $\langle e_1, e_2\rangle \in \hat{G}_\rho.\mathsf{po}$.

In addition, $(\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl} \subseteq C$ ensures that there is a pair $\langle S_\sigma, C_\sigma\rangle$ that satisfies (II), (V), (VI), (VII) of Definition 10 and $C_\sigma \subseteq C$. Let $S_\sigma = \delta(\hat{G}_\rho.\mathsf{Evts})$ and $C_\sigma = G_\sigma.((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl}) \cap \delta(\hat{G}_\rho.\mathsf{Evts})$.

II S_σ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$. This can be shown by noticing that $\hat{G}_\rho.\mathsf{Evts}$ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$.

V For each $e \in S_\sigma$, if $\langle r, e\rangle \in G_\sigma.\mathtt{ctrl}$, then $r \in C_\sigma$. This can be shown by noticing $\mathtt{ctrl} \subseteq ((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+$.

VI For each $e \in C_\sigma$, if $\langle r, e\rangle \in G_\sigma.\mathtt{data}$, then $r \in C_\sigma$. By definition of $\mathtt{data}$, we know that e is a write. Then for $e \in C_\sigma$, there must be a read $r' \in C_\sigma$ such that $e \xrightarrow{\mathtt{rf}} r'$. Given that $(\mathtt{data}; \mathtt{rf})^*; \mathtt{ctrl} \subseteq ((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+$, we have $r \in C_\sigma$.

VII For each $r \in C_\sigma$, there exists a write $w \in C_\sigma$ such that $r.val = w.val$ and $r.loc = w.loc$. This is ensured by the well-formedness of $G_\sigma.\mathtt{rf}$.

The similar reasoning steps as in the proof of Proposition 5, (II), (V), and (VI) ensures executability of $\hat{G}_\rho$. (VII) ensures that $\hat{G}_\rho$ is read-feasible.

□

▶ Lemma 6
Let $\hat{G}_\rho = (\mathsf{Evts}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evts}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{sync})$, then $\hat{G}_\rho$ is lock-feasible.

Proof. Suppose, towards contradiction, that $\hat{G}_\rho$ is not lock feasible. Then there exists a lock l such that there are two open critical sections in $\hat{G}_\rho$ with acquire events a_1 and a_2. Then there are release events $r_1 = \mathsf{match}(a_1)$ and $r_2 = \mathsf{match}(a_2)$ in G_σ such that either $r_1 \xrightarrow{\mathtt{sync}} a_2$ or $r_2 \xrightarrow{\mathtt{sync}} a_1$ and neither of them are included in $\hat{G}_\rho$. This immediately contradicts with the hypothesis that $\hat{G}_\rho$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{sync})$, which implies that at least one of r_1 and r_2 is included in $\hat{G}_\rho$. Thus, $\hat{G}_\rho$ is lock-feasible. □

▶ Lemma 7
Let G_σ be an input execution such that G_σ is sequentially consistent, equipped with a linear $\mathtt{trace}$ order. Let $\hat{G}_\rho$ be a symbolic plain execution that is well-formed, and $\hat{G}_\rho \rhd b$ where b is a reported bug. If for all acquire event $\mathsf{acq}(l) \in \hat{G}_\rho.\mathsf{Evts}$ such that $l \in \mathsf{LocksHeld}(e)$, $\langle \mathsf{acq}(l), e \rangle \in G_\sigma.\mathtt{trace}$ for each event $e \in b.\mathsf{Evts}$ that is in a critical section where the acquire event of the critical section $\mathsf{acq}(l) \in \hat{G}_\rho$, then there exists a memory order insertion scheme over $\hat{G}_\rho.\mathsf{Evts}$ such that $\hat{G}_\rho$ is sequentially consistent.

Proof. Set

$$\hat{G}_\rho.\mathtt{rf} = G_\sigma.\mathtt{rf}|_C$$

$$\hat{G}_\rho.\mathtt{co} = \delta^{-1}(G_\sigma.\mathtt{co}|_{\delta(\hat{G}_\rho.\mathsf{Wrts})})$$

$$\hat{G}_\rho.\mathtt{sync} = G_\sigma.\mathtt{sync} \cap (\hat{G}_\rho.\mathsf{Rels} \times \mathsf{Acqs})$$

where C is the set of concrete events in $\hat{G}_\rho$. Since $\hat{G}_\rho$ is read feasible, $\hat{G}_\rho.\mathtt{rf}$ is well-formed over concrete events.

Since G_σ is sequentially consistent, the inserted orders in $\hat{G}_\rho$ do not form a $(\mathsf{po} \cup \mathtt{sync} \cup \mathtt{com})^+$ cycle.

For each $e \in b.\mathsf{Evts}$ such that e is in a critical section, let the acquire event of the critical section be $\mathsf{acq}(l) \in \hat{G}_\rho.\mathsf{Acqs}$. Then this critical section is an open critical section in $\hat{G}_\rho$. Since $\hat{G}_\rho$ is lock feasible, all other critical sections of l are closed, i.e., each $\mathsf{acq}(l)' \in \hat{G}_\rho.\mathsf{Acqs}$ has a matching $\mathsf{rel}(l)'$. Since $\hat{G}_\rho.\mathtt{sync} = G_\sigma.\mathtt{sync} \subseteq G_\sigma.\mathtt{trace}$, we have $\langle \mathsf{rel}(l)', \mathsf{acq}(l)\rangle \in \hat{G}_\rho.\mathtt{sync}$ for all $\mathsf{rel}(l)' \neq \mathsf{match}(\mathsf{acq}(l))$. Hence, all open critical sections are ordered last in $\hat{G}_\rho$.

With both requirements satisfied, we can conclude that $\hat{G}_\rho$ is sequentially consistent. □

▶ LEMMA 8
Let $\hat{G}_\rho$ be a symbolic execution with a well-formed $\mathtt{rf}$-map over concrete events, a total $\mathtt{co}$ order over write events to the same location, and a well-formed $\mathtt{sync}$ over lock events. If $\hat{G}_\rho$ is $\mathcal{M}$-consistent and $\hat{G}_\rho \in [\![P]\!]$ with $e.val \in \mathsf{Val}$ for each $e \in \mathsf{preserve}(b)$, then there exists a map $\Theta : \mathsf{Sym} \rightarrow \mathsf{Val}$ such that the concrete execution $G_\rho = \Theta(\hat{G}_\rho)$ with a complete $\mathtt{rf}$-map is $\mathcal{M}$-consistent and $G_\rho \in [\![P]\!]$.

Proof. We begin by inserting $\mathtt{rf}$ orders so that for each (symbolic) read event $r \in \hat{G}_\rho.\mathsf{Rds}$, there is a unique write event $w \in \hat{G}_\rho.\mathsf{Wrts}$ such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{rf}$ and $w.loc = r.loc$. In the rest of the proof, we define the global happens-before as $\mathtt{ghb} = \mathtt{com} \cup \mathtt{sync} \cup \mathtt{ppo} \cup (\mathtt{po}; [L]) \cup ([L]; \mathtt{po})$.

Claim: Let $t \in \hat{G}_\rho.\mathsf{Thrd}$ and $r \in \hat{G}_\rho.\mathsf{Rds}$ be the first read event with symbolic value in thread t. Let $\hat{G}'_\rho$ be the resulting graph after inserting $\langle w, r\rangle$ to $\hat{G}_\rho.\mathtt{rf}$, where w is the $\mathtt{co}$-maximal write such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{ghb}$. If there is no such write event, then w is the initial write to the memory location $r.loc$. Then $\hat{G}'_\rho$ is still $\mathcal{M}$-consistent.

Proof of Claim: Comparing to $\hat{G}_\rho$, $\hat{G}'_\rho$ has the following memory orders inserted: $\langle w, r\rangle \in \mathtt{rf}$, and $\langle r, w'\rangle \in \mathtt{fr}$ for all w' such that $\langle w, w'\rangle \in \hat{G}_\rho.\mathtt{co}$. Since w is the $\mathtt{co}$-maximal write event such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{ghb}$, we know that $\langle w', r\rangle \notin \hat{G}_\rho.\mathtt{ghb}$. Hence, there is no new $\mathtt{ghb}$ cycle formed with the newly inserted orders. Hence, $\hat{G}'_\rho$ is also $\mathcal{M}$-consistent.

For each thread $t \in \hat{G}_\rho.\mathsf{Thrd}$, we start from the beginning of t and insert $\mathtt{rf}$ orders for each symbolic read events according to the previous claim. Since $\mathcal{M}$-consistency is maintained at each step, the resulting execution graph is $\mathcal{M}$-consistent.

Let $\hat{G}'_\rho$ be the resulting execution graph after the previous step of inserting $\texttt{rf}$ orders. We now show that there is a map $\Theta : \mathsf{Sym} \to \mathsf{Val}$ that maps $\hat{G}'_\rho$ to a concrete execution G_ρ.

Claim: For each event $e \in \hat{G}'_\rho$ such that $e.val \in \mathsf{Sym}$, then there exists a value $v \in \mathsf{Val}$ such that the resulting execution graph $\hat{G}''_\rho$ after substituting $e.val$ by v is executable, i.e., $\hat{G}''_\rho \in [\![P]\!]$, and well-formed.

Proof of Claim: Induction on the events on $(\texttt{rf} \cup \texttt{data})^+$ chain. Let $e \in \hat{G}'_\rho.\mathsf{Evts}$. If there is no event $(\texttt{rf} \cup \texttt{data})^+$-ordered before e or all events $(\texttt{rf} \cup \texttt{data})^+$-ordered before e are concrete, then set $e.val = \delta(e).val$. Then obviously the resulting execution graph $\hat{G}''_\rho \in [\![P]\!]$.

If $e \in \hat{G}'_\rho.\mathsf{Rds}$, then there exists a unique write event $w \in \hat{G}'_\rho.\mathsf{Wrts}$ such that $\langle w, e\rangle \in \hat{G}'_\rho.\texttt{rf}$. If $w.val \in \mathsf{Val}$, then set $e.val = w.val$. If $w.val \in \mathsf{Sym}$, then by induction hypothesis, there exists a value $v \in \mathsf{Val}$ for w such that the resulting execution graph is executable and well-formed. Set $e.val = v$. The resulting graph is executable because $e \notin C_\sigma$, and well-formed since $e.val = w.val$.

If $e \in \hat{G}'_\rho.\mathsf{Wrts}$, since $\hat{G}'_\rho \in [\![P]\!]$, there exists a state st_i right before e is generated such that $st_i.sprog(pc) = [x] \texttt{ := } expr$. By induction hypothesis, for each read event r such that $\langle \delta(r), \delta(e)\rangle \in G_\sigma.\texttt{data}$, there exists a value such that the resulting graph is executable and well-formed. Set $e.val = subst(st_i.\Phi^\theta(expr), val(st_i.\Psi(expr)))$ where $val : \mathsf{Sym} \to \mathsf{Val}$ is a value map for those reads. The resulting graph is executable and well-formed because $e \notin C_\sigma$ and each read that e depends on is mapped to a concrete value before $e.val$ is mapped.

Then G_ρ can be obtained by following the $(\texttt{rf} \cup \texttt{data})^+$ chain and replace the value of each symbolic event by a concrete value according to the above description. By previous claims, G_ρ is $\mathcal{M}$-consistent, executable, and well-formed.

□

APPENDIX P

Enhanced-MCR-tso

In this section, we define an enhanced version of the previous algorithm that uses a novel idea of *transformation* to predict data races under the TSO model. This enables us to catch more data races than the work of [HH16]. We provide an example in Appendix R.

In addition to the constraints defined above, we preprocess the input execution G_σ with *read elimination*. During this phase, all removable read events are eliminated from the trace, where removable(G_σ) is a subset of read events that can be found inductively as described as the following.

For each read event $r \in G_\sigma.\mathsf{Rds}$, if there is a write event $w \in G_\sigma.\mathsf{Wrts}$ such that $\langle w, r\rangle \in G_\sigma.(\mathtt{rf} \cap \mathsf{po})$, then $r \in$ removable(G_σ) if there is no event such that $w \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} r$, or for each event e such that $w \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} r$, either e is a write event where $e.loc \neq r.loc$, or $e \in$ removable(G_σ).

After the read events in removable(G_σ) are removed from the execution, we use the same set of constraints, $\Phi_{\mathsf{ppo}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}$, to determine if $\langle e_1, e_2\rangle$ forms a data race.

Before we start proving its soundness, we introduce a proposition from the work of [LV16].

▶ Proposition 12
If $G \rightsquigarrow_{tso} G'$ and G' is TSO-consistent, then G is TSO-consistent.

In [LV16], two transformations were considered, read elimination and write-read reordering. $G \rightsquigarrow_{tso} G'$ iff G' can be obtained via either of the transformations. Therefore, we can use this proposition for read elimination.

The soundness theorem is the following.

▶ THEOREM 26
If there exists a map $\rho : \mathcal{O} \to \mathsf{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\mathsf{ppo}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}(e_1, e_2)$ after read elimination, then $\langle e_1, e_2 \rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a lock-feasible event set that is downward-closed w.r.t. $G_\sigma.\mathsf{po} \cup \mathtt{rf}$ from $b = e_1 e_2$.Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\mathsf{Evts} \to S_\sigma.\mathsf{Evts}$ such that $\delta(e) \approx e$. Let $C = G_\sigma.\mathsf{Wrts} \cup \mathsf{Rds}$. If $\delta(e) \in C$, then we set $\delta(e) = e$. Let $\hat{G}_\rho$ be a symbolic plain execution graph such that $\hat{G}_\rho.\mathsf{Evts} = S_\rho$ and $\hat{G}_\rho.\mathsf{po} = \delta(G_\sigma.\mathsf{po}|_{S_\sigma})$.

By LEMMA 5 and the fact that S_σ is lock-feasible, we get $\hat{G}_\rho$ is well-formed and executable. In addition, since S_σ is downward-closed from b, $\hat{G}_\rho \rhd b$.

▶ *Inserting Memory Orders.* Note that not all events in $\hat{G}_\rho$ is assigned to a number by ρ, due to the read elimination transformation. Let $\hat{G}_\rho{}'$ be the symbolic execution after applying read elimination on $\hat{G}_\rho$. Then every event in $\hat{G}_\rho{}'$ is assigned to a natural number by ρ. By the same order insertion scheme as in the proof of THEOREM 6, we have that $\hat{G}_\rho{}'$ is TSO-consistent. By PROPOSITION 12, $\hat{G}_\rho$ is also TSO-consistent.

▶ *Mapping to Concrete Execution Graph.* Lastly, by LEMMA 8, there is a concrete execution graph that inherit all the properties above. □

APPENDIX Q

A TSO Race Predictable from SC trace

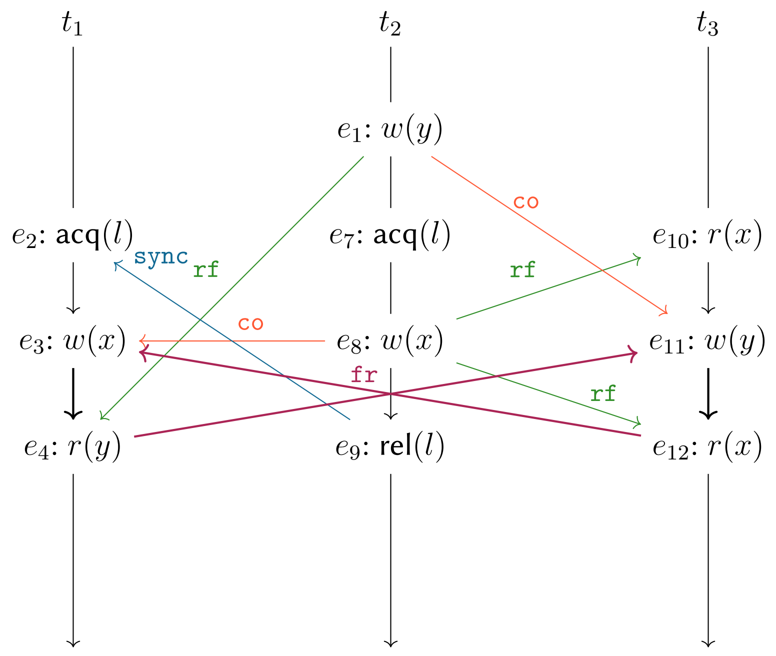
In this section, we provide an example of a TSO data race that is discoverable by MCR-TSO [HH16]. The example was originally from the paper by [Pav20, Figure 8] to show a non-SC-race.

The input execution traces are represented in the form shown in Fig. Q.1a. Each column represents a thread and each row represents a time-stamp. At each time-stamp, there is exactly one event being executed. Events from different threads can be executed in a interleaving style while respecting the mutual exclusion property of locks. We use e_i to identify each event, where i is the time-stamp when e_i is executed. There are four types of events: read, write, lock acquire, and lock release. We write $r(x)$ and $w(x)$ for a read and a write event on a memory location x, and acq(l) and rel(l) for an acquire and a release event on a lock l. The highlighted events are reported as data races. In Fig. Q.1a, the two highlighted events, e_5 and e_{13} is not a data race under sequential consistency. Indeed,

- If e_5 and e_{13} is a data race, then Fig. Q.1b shows all the events that have to occur before e_5 and e_{13}. The order among these events has to respect the program order. Therefore, the order of events on each thread follows the same order as captured in the input trace from Q.1a.
- M2 requires each read event to read from the same write event as they appeared in the input trace to maintain soundness. Therefore, we have $e_8 \to e_{10}$ and $e_8 \to e_{12}$ for location x, and $e_1 \to e_4$ for location y.
- Since e_5 is in a critical section protected by lock l, its critical section has to be ordered after all other critical sections protected by the same lock. In this example, it means the critical section

	t_1	t_2	t_3
1		$w(y)$	
2	acq(l)		
3	$w(x)$		
4	$r(y)$		
5	**w(z)**		
6	rel(l)		
7		acq(l)	
8		$w(x)$	
9		rel(l)	
10			$r(x)$
11			$w(y)$
12			$r(x)$
13			**r(z)**

(a) The Input Trace



(b) A TSO-Consistent Witness Execution
Figure Q.1: Example from Fig. 8 in [Pav20]

on t_2 has to be ordered before the critical section on t_1. Hence, we can infer $e_9 \to e_2$.

- By the transitivity of the partial order constructed so far, we can see that $e_8 \to e_3$. Since there are two read events on t_3 reading from e_8, then they must be ordered before e_3. Therefore, $e_{12} \to e_3$.
- Since $e_8 \to e_{10}$, by the transitivity of the partial order again, we have $e_1 \to e_{11}$. Since e_4 reads from e_1, then it must be that $e_4 \to e_{11}$.
- But now a cycle occurs: $e_3 \to e_4 \to e_{11} \to e_{12} \to e_3$.

On the other hand, the cycle derived in Fig. Q.1b is allowed under TSO [OSS09]. Indeed, Fig. Q.1b also shows the execution with orders among events augmented with specific semantics defined by the axiomatic memory model of TSO. The po order $e_3 \to e_4$ and $e_{11} \to e_{12}$ are not preserved program order. As a result, the po$\cup$`fr` cycle shown in the figure is allowed by the TSO model. Therefore, we can conclude that e_5 and e_{13} form a data race under TSO.

Note that the constraint encoding of MCR can capture this data race, precisely because the constraints do not require e_3 to be ordered before e_4 and e_{11} to be ordered before e_{12}. As a result, there exists a satisfiable solution of the event orders.

APPENDIX R

TSO Race Discovered by Read Elimination

In §4.6, we showed an enhanced version of a data race predictor augmented with Read Elimination. In this section, we show an example that exhibit a sound TSO data race that *cannot* be caught by the TSO analysis by [HH16], but can be caught by our enhanced race predictor.

Fig. R.1a shows a sequential trace of four threads. The highlighted events, e_8 and e_{17}, form a data race under the TSO [OSS09] model. However, the constraints from [HH16] fail to catch this data race due to overly strong restriction imposed by Φ_{mem}. The constraints on memory events, Φ_{mem} is defined as

$$\Phi_{mem} = \Phi_{ww} \wedge \Phi_{rr} \wedge \Phi_{rw} \wedge \Phi_{addr}$$

where Φ_{ww} enforces the write-write program orders, Φ_{rr} enforces the read-read program orders, Φ_{rw} enforces the read-write program orders, and Φ_{addr} enforces the program orders between memory events accessing the same location. Note that the first three constraints enforce the preserved program orders (ppo) of TSO and the last constraint enforces the program order restricted to the same location (`po-loc`). The constraint includes all four of them in *one* transitive partial order, while the TSO model considers them *separately* in two assertions:

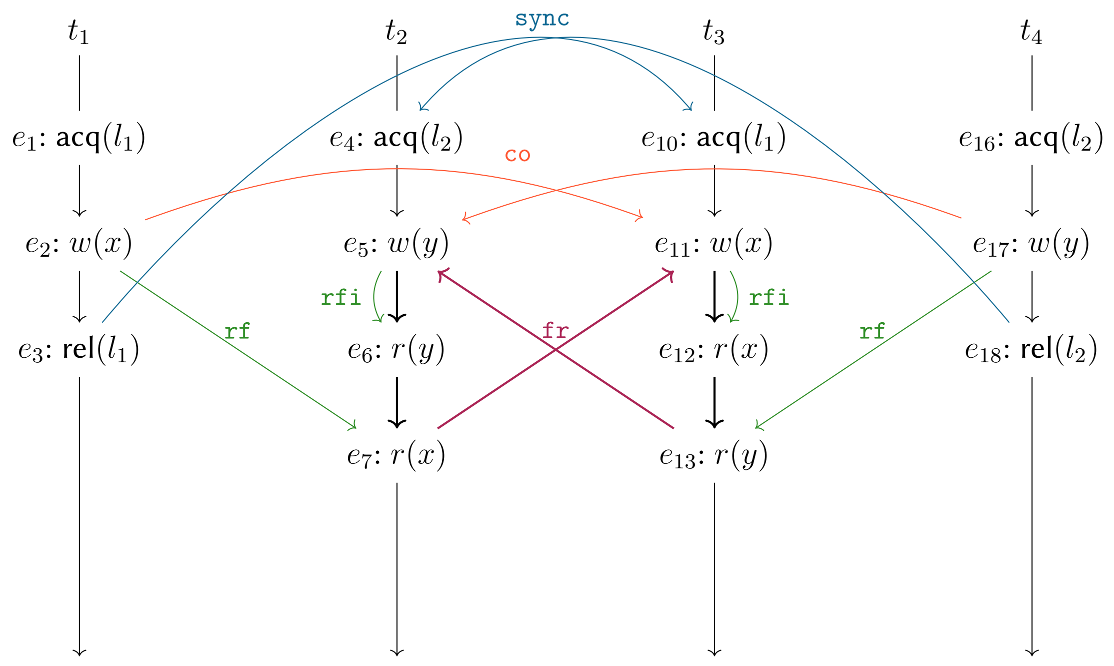
$$\text{irreflexive } (\texttt{po-loc} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})^{+}$$

$$\text{irreflexive } (\texttt{ppo} \cup \texttt{rfe} \cup \texttt{fr} \cup \texttt{co})^{+}$$

In the example, the program orders $e_5 \rightarrow e_6$ and $e_{11} \rightarrow e_{12}$ are `po-loc` orders whereas $e_6 \rightarrow e_7$

	t_1	t_2	t_3	t_4
1	$\mathsf{acq}(l_1)$			
2	$w(x)$			
3	$\mathsf{rel}(l_1)$			
4		$\mathsf{acq}(l_2)$		
5		$w(y)$		
6		$r(y)$		
7		$r(x)$		
8		**w(z)**		
9		$\mathsf{rel}(l_2)$		
10			$\mathsf{acq}(l_2)$	
11			$w(y)$	
12			$\mathsf{rel}(l_2)$	
13				$\mathsf{acq}(l_1)$
14				$w(x)$
15				$r(x)$
16				$r(y)$
17				**r(z)**
18				$\mathsf{rel}(l_1)$

(a) The Input Trace



t_1
t_2
t_3
t_4
sync
co
e_1: $\mathsf{acq}(l_1)$
e_4: $\mathsf{acq}(l_2)$
e_{10}: $\mathsf{acq}(l_1)$
e_{16}: $\mathsf{acq}(l_2)$
e_2: $w(x)$
e_5: $w(y)$
e_{11}: $w(x)$
e_{17}: $w(y)$
rfi
rfi
rf
fr
rf
e_3: $\mathsf{rel}(l_1)$
e_6: $r(y)$
e_{12}: $r(x)$
e_{18}: $\mathsf{rel}(l_2)$
e_7: $r(x)$
e_{13}: $r(y)$

(b) Witness Execution

Figure R.1: A TSO Race Captured by Race Predictor after Read Elimination

and $e_{12} \rightarrow e_{13}$ are ppo orders. Together with the fr edges from e_7 to e_{11} and from e_{13} to e_5, they form a cycle, as shown in Fig. R.1b. The cycle makes the constraints unsatisfiable, but is allowed under the memory model of TSO.

On the other hand, augmenting the constraints with the Read Elimination transformation suffices to cover this case. Note that e_6 and e_{12} are removed after the Read Elimination trasformation. The resulting execution graph corresponds to a standard store-buffering (SB) litmus test, which can be captured by the constraint Φ_{mem}.

APPENDIX S

WRC-race: another example

In this section, we provide another example of data race under weak memory that can be predicted from an SC trace while preserving the same observable behavior. Fig. S.1a shows a sequential trace of four threads. The highlighted events, e_3 and e_{12}, form a data race under the ARMv8 [ADG21] model, assuming there is no data or address dependencies among the events and no control dependency between e_{10} and e_{11}.

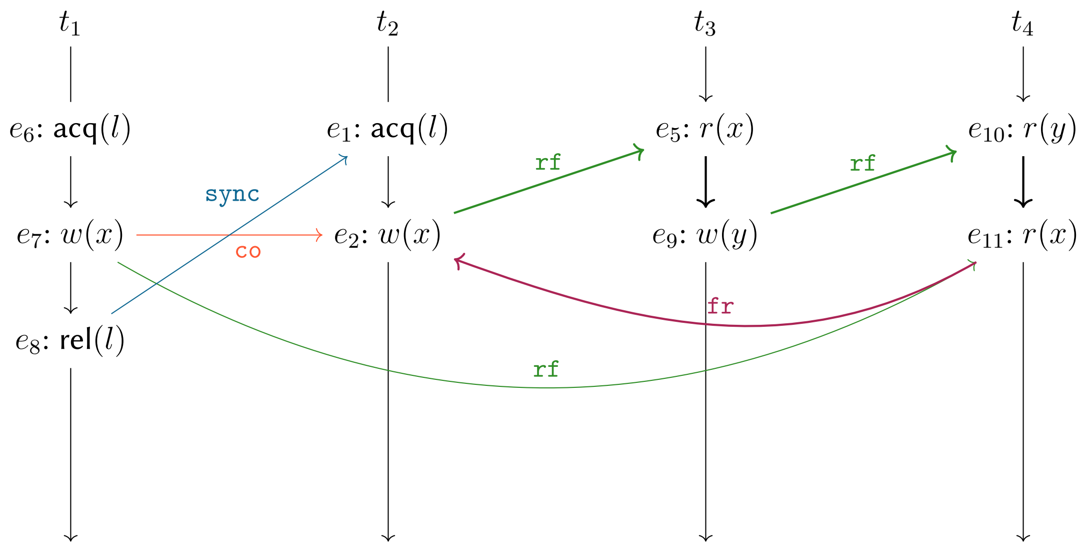
First, note that e_3 and e_{12} is not an SC race. To see this, following steps similar to M2 [Pav20], we have:

- All events except for the release in t_2 (e_4) are included in a witness execution.
- Since we have an open acquire in t_2, the critical section in t_2 should be ordered after the critical section in t_1, yielding a coherence order from $w(x)$ at line 7 to $w(x)$ at line 2.
- From the inferred coherence order, we get $r(x)$ at line 11 is ordered before $w(x)$ at line 2 (since it reads from the write at line 7)
- Now we have a cycle: $e_2 : w(x) \xrightarrow{\texttt{rf}} e_5 : r(x) \xrightarrow{\texttt{po}} e_9 : w(y) \xrightarrow{\texttt{rf}} e_{10} : r(y) \xrightarrow{\texttt{po}} e_{11} : r(x) \xrightarrow{\texttt{fr}} e_2 : w(y)$. The cycle is highlighted in Fig. S.1b.

On the other hand, the cycle is allowed under the ARMv8 [ADG21] model and the witness execution in Fig. S.1b is consistent. To see this, note that the program order $e_{10} \rightarrow e_{11}$ is not included in the *locally-ordered-before* order and hence the external visibility requirement of the ARMv8 model is satisfied.

	t_1	t_2	t_3	t_4
1		acq(l)		
2		$w(x)$		
3		**w(z)**		
4		rel(l)		
5			$r(x)$	
6	acq(l)			
7	$w(x)$			
8	rel(l)			
9			$w(y)$	
10				$r(y)$
11				$r(x)$
12				**r(z)**

(a) The Input Trace



(b) Witness Execution

Figure S.1: A Data Race under the ARMv8 model

APPENDIX T

Program Transformations

T.1 Deordering and Reordering

▶ Theorem 12 (Deordering)
Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a deordering operation on a pair of accesses a and b according to Fig. 5.3. Let G_{tgt} be an execution of P_{tgt}. Then there exists an execution G_{src} of P_{src} such that

- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\}$ where a and b are po-adjacent
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{src}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$

and if G_{tgt} is Jam21-consistent, then G_{src} is Jam21-consistent.

Proof. Note that for the Coherence requirement, only three kinds of edges contributes to co-jam: vo, rf, and po to the same location. Since here we are considering deorderable pairs, which are pairs of accesses to different locations related by po in G_{src}, we only need to consider whether deordering them would affect the set of vo in the execution. We can analyze this case by case.

- $(r_x * r_y)$. The No-Thin-Air requirement is satisfied automatically in G_{src} since we are deordering a pair of reads. Since $\mathsf{po} \cap (r_x^{\sqsubseteq \mathsf{Opq}} * r_y) \not\subseteq \mathtt{vo}$, it follows that G_{src} is also Jam21-consistent.

- $(r_x * w_y)$. G_{src} fulfills COHERENCE since the po edge between two accesses whose access mode is weaker or equal to Opaque mode does not contribute to any new `vo` edge. In addition, G_{src} fulfills the NO-THIN-AIR requirement because one of the accesses is in Plain mode where as NO-THIN-AIR only requires the acyclicy of po $\cup$ `rf` among Opaque mode accesses.

- $(r_x * rmw_y)$. First note that the Volatile mode for read-modify-write events include the effect of Release Mode. G_{src} fulfills COHERENCE since the po edge between r_x and rmw_y does not contribute to any new `vo` edge. G_{src} fulfills NO-THIN-AIR because r_x is Plain mode.

- $(r_x * F)$. It's easy to see that the po edge added in G_{src} does not contribute to any new `vo` edge therefore the COHERENCE is satisfied. Since we are deordering a read and a fence, G_{src} fulfills NO-THIN-AIR automatically.

- $(w_x * r_y)$. It's easy to see that the po edge added in G_{src} does not contribute to any new `vo` edge therefore the COHERENCE is satisfied. Since we are deordering a write and a read, G_{src} fulfills NO-THIN-AIR automatically. The only situation the two accesses cannot be deordered is when they are both in Volatile mode because the po between two Volatile accesses can be derived into a `vo` order.

- $(w_x * w_y)$. It's easy to see that the po edge added in G_{src} does not contribute to any new `vo` edge therefore the COHERENCE is satisfied. Since we are deordering a write and a write, G_{src} fulfills NO-THIN-AIR automatically. Here we need the second write w_y to be weaker than release mode to ensure that the po between the two accesses does not contribute to the `vo` order.

- $(w_x * rmw_y)$. It's easy to see that the po edge added in G_{src} does not contribute to any new `vo` edge therefore the COHERENCE is satisfied. Since we are deordering a write and a read-modify-write, G_{src} fulfills NO-THIN-AIR automatically. Since rmw_y is both in the set of reads and in the set of writes of the execution graph G_{src}, we take the intersection of previous cases. In addition, `rel` and `acq` do not subsume each other, so it is safe for o_2 to be `acq` mode.

- $(w_x * F)$. It's easy to see that the po edge added in G_{src} does not contribute to any new `vo` edge therefore the COHERENCE is satisfied. Since we are deordering a write and a fence, G_{src} fulfills NO-THIN-AIR automatically. Here we are basically avoiding the situation when w_x and F can form any svo. In addition, there are also situations where w_x and F have a svo if the writes that follows the fence are already in rel modes. Since svo and ra are considered equivalently in terms of their memory order effect in the JAM21 model, the svo they form is redundant in the presence of all the ra.

- $(rmw_x * r_y)$. It's easy to see that the po edge added in G_{src} does not contribute to any new vo edge therefore the COHERENCE is satisfied. Since we are deordering a read-modify-write and a read, G_{src} fulfills NO-THIN-AIR automatically. Here we want to avoid the rmw_x to have an access mode stronger or equal to acq mode because it'd create an ra edge which is considered as a vo edge.
- $(rmw_x * w_y)$. It's easy to see that the po edge added in G_{src} does not contribute to any new vo edge therefore the COHERENCE is satisfied. For the NO-THIN-AIR requirement, since w_y is in Plain mode, it does not contribute to any po $\cup$ rf cycle among Opaque accesses.
- $(rmw_x * rmw_y)$. They cannot be deordered due to the NO-THIN-AIR requirement. (Note that read-modify-write operations are atomic by definition, so there is no Plain mode or Opaque mode for RMW events).
- $(rmw_x * F)$. Similar to the previous cases.
- Deordering with fence. Similar to the previous cases.

□

▶ COROLLARY 2 (REORDERING)
JAM21 supports the reordering transformation for pairs of adjacent accesses shown in Fig. 5.3.

Proof. Let a and b be a pair of such memory events and $a \xrightarrow{\text{po}} b$ in G_{src}. By Theorem 12, we know that removing the po edge between a and b does not introduce new program behavior. Let

G' be the execution graph after the deordering transformation. By Theorem 11, we know that adding a po edge from b to a in G' does not introduce new program behavior either. Therefore, reordering of access pairs in Fig. 5.3 is supported by JAM21. $\square$

T.2 Merging

T.2.1 Read-read Merging

▶ THEOREM 13 (READ-READ MERGING)
Let G_{tgt} be an JAM21-consistent execution. Let $a \in G_{tgt}.\mathsf{Rds}\backslash\mathtt{RMW}$ and let $a' \in G_{tgt}.\mathsf{Evts}$ such that $a \xrightarrow{\mathtt{rf}} a'$. Let $b \notin G_{tgt}.\mathsf{Evts}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a', b\rangle\}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $b \in G_{src}.\mathsf{Rds}$
- $G_{src}.AccessMode(b) = G_{src}.AccessMode(a) \sqsubseteq \mathsf{Acquire}$

and G_{src} is JAM21-consistent.

Proof. We show that G_{src} fulfills the two requirements needed to be JAM21-consistent.

- Suppose G_{src} violates the NO-THIN-AIR requirement, then there is a $(\mathsf{po} \cup \mathtt{rf})+$ cycle involving b. If we have $a' \xrightarrow{\mathtt{rf}} b \xrightarrow{(\mathsf{po}|\mathtt{rf})+} a'$, then $a' \xrightarrow{\mathtt{rf}} a \xrightarrow{(\mathsf{po}|\mathtt{rf})+} a'$. If we have $a \xrightarrow{\mathsf{po}} b \xrightarrow{(\mathsf{po}|\mathtt{rf})+} a$, then $a \xrightarrow{(\mathsf{po}|\mathtt{rf})+} a$. In both of the cases, G_{tgt} is inconsistent, which contradict with our previous assumption. Therefore, the NO-THIN-AIR requirement is satisfied by G_{src}.
- Suppose G_{src} violates the COHERENCE requirement, then there is a $\mathtt{co}$ cycle. Note that $AccessMode(b) =$

$AccessMode(a) \sqsubseteq$ Acq. In addition, for all events i, if $b \xrightarrow{\texttt{vo}} i$, then $a \xrightarrow{\texttt{vo}} i$ and for all events j, if $j \xrightarrow{\texttt{vo}} b$, then $j \xrightarrow{\texttt{vo}} a$. Therefore, for any coherence cycle derived from the edges from and to b, there is also a coherence cycle derived from the edges from and to a. If G_{src} has a co cycle, G_{tgt} also has a co cycle, which contradicts with our previous assumption.

$\square$

Counter Example Here, we give an example showing that read-read merging is not allowed by JAM21 if the read accesses are both Volatile mode. Consider the following program:

```
Thread0 {
int r1 = X.getOpaque(); // 1
int r2 = X.getOpaque(); // 2
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }

Thread2 {
    X.setOpaque(2);
  }
```

```
Thread3 {
int r5 = X.getVolatile(); // 2
int r6 = X.getVolatile(); // 2
Y.setRelease(1);
}

Thread4 {
    Y.setVolatile(2);
    X.setVolatile(1);
  }
```

Applying the read-read merging transformation to this program yields:

```
Thread0 {
int r1 = X.getOpaque(); // 1
int r2 = X.getOpaque(); // 2
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }

Thread2 {
    X.setOpaque(2);
  }
```

```
Thread3 {
int r5 = X.getVolatile(); // 2
int r6 = r5
Y.setRelease(1);
}

Thread4 {
    Y.setVolatile(2);
    X.setVolatile(1);
  }
```

The execution graphs with the annotated read values is shown in Fig. T.1 and Fig. T.2.

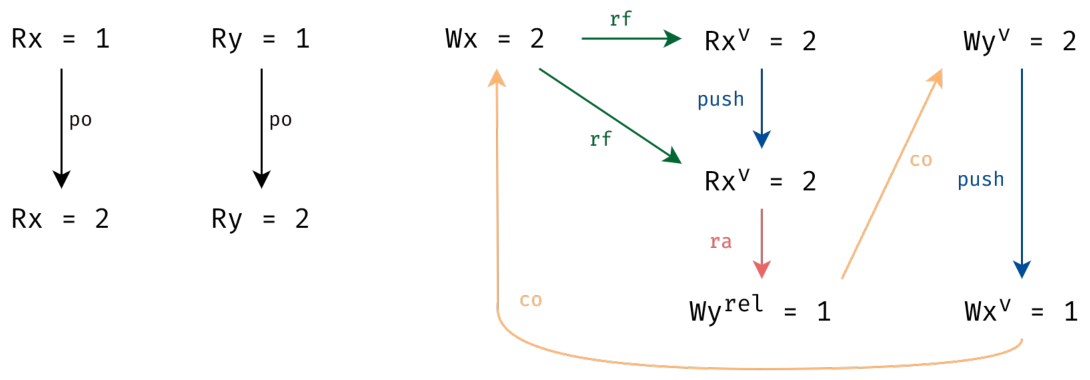


Figure T.1: Execution Graph before read-read merge on Volatile (Forbidden)

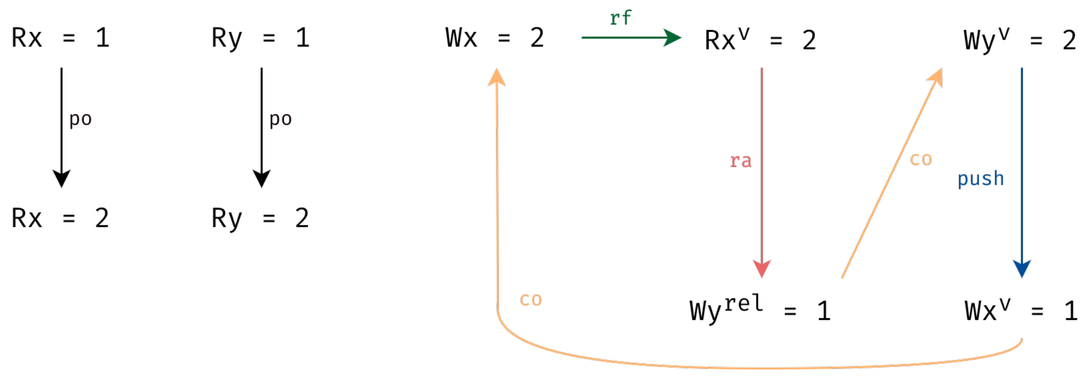


Figure T.2: Execution Graph after read-read merge on Volatile (Allowed)

For the two read accesses of x on Thread 3, one may think it's OK to merge them into one. However, since they are Volatile accesses, they also impose a push edge which is totally ordered with other push edges. Merging the two reads removes the synchronization provided by the push edge, introducing the program behavior shown in Fig. T.2.

T.2.2 Write-write Merging

▶ Theorem 14 (Write-Write Merging)
Let G_{tgt} be an Jam21-consistent execution. Let $b \in G_{tgt}.\text{Wrts}\backslash\texttt{RMW}$ and let $a \notin G_{tgt}.\text{Evts}$ and $loc(a) = loc(b) \wedge \forall i \in G_{tgt}.\text{Wrts}, loc(i) = loc(b) \Rightarrow val(a) \neq val(i)$. There exists a G_{src} such

that:

- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf}$
- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{a\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $a \in G_{src}.\mathsf{Wrts}$
- $G_{src}.AccessMode(a) = G_{src}.AccessMode(b) \sqsubseteq \mathsf{Release}$

and G_{src} is JAM21-consistent.

Proof. We show that G_{src} fulfills the two requirements to be JAM21-consistent.

- NO-THIN-AIR. Note that $a \xrightarrow{\mathtt{po}} b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$ implies that $b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} b$. Therefore, if G_{src} violates NO-THIN-AIR, G_{tgt} also violates NO-THIN-AIR, contradicting to our previous assumption.
- COHERENCE. First note that there is no extra $\mathtt{rf}$ edge from a and $\forall i, (i \xrightarrow{\mathtt{vo}} a \Rightarrow i \xrightarrow{\mathtt{vo}} b) \wedge (a \xrightarrow{\mathtt{vo}} i \Rightarrow b \xrightarrow{\mathtt{vo}} i)$ (because a and b have the same access mode and they are not in Volatile mode). Therefore, any $\mathtt{co}$ cycle derived from a, we can derive the same $\mathtt{co}$ cycle with b. While $a \xrightarrow{\mathtt{po}} b$ implies that $a \xrightarrow{\mathtt{vo}} b$, since there is no $\mathtt{rf}$ edge from a, it cannot contribute to any extra $\mathtt{co}$ cycle. Therefore, if there is a $\mathtt{co}$ cycle in G_{src}, then it implies that there is a $\mathtt{co}$ cycle in G_{tgt}, contradicting to our previous assumption.

□

Counter Example We now provide a counter-example showing write-write merge is not valid for Volatile mode writes. Consider the following example program:

```
Thread0 {
int r1 = X.getOpaque(); // 2
int r2 = X.getOpaque(); // 3
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }
```

```
Thread2 {
Y.setOpaque(2);
X.setVolatile(1);
X.setVolatile(2);
}

Thread3 {
    X.setVolatile(3);
    Y.setVolatile(1);
  }
```

The execution graph of the program before the transformation is shown in Fig. T.3.

Applying write-write merging transformation to Thread 2, we have:

```
Thread0 {
int r1 = X.getOpaque(); // 2
int r2 = X.getOpaque(); // 3
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }
```

```
Thread2 {
Y.setOpaque(2);
X.setVolatile(2);
}

Thread3 {
    X.setVolatile(3);
    Y.setVolatile(1);
  }
```

The execution graph after the transformation is shown in Fig. T.4. After removing the write access in Volatile mode, the cross-thread synchronization effect between Thread 2 and Thread 3 is also removed, introducing the new behavior in the figure.

T.2.3 Write/RMW-read Merging

▶ THEOREM 15 (WRITE/RMW-READ MERGING)
Let G_{tgt} be a JAM21-consistent execution. Let $a \in G_{tgt}.\mathsf{Wrts}$ and $b \notin G_{tgt}.\mathsf{Evts}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $b \in G_{src}.\mathsf{Rds}$
- $G_{src}.loc(b) = G_{src}.loc(a)$
- $G_{src}.val(b) = G_{src}.val(a)$

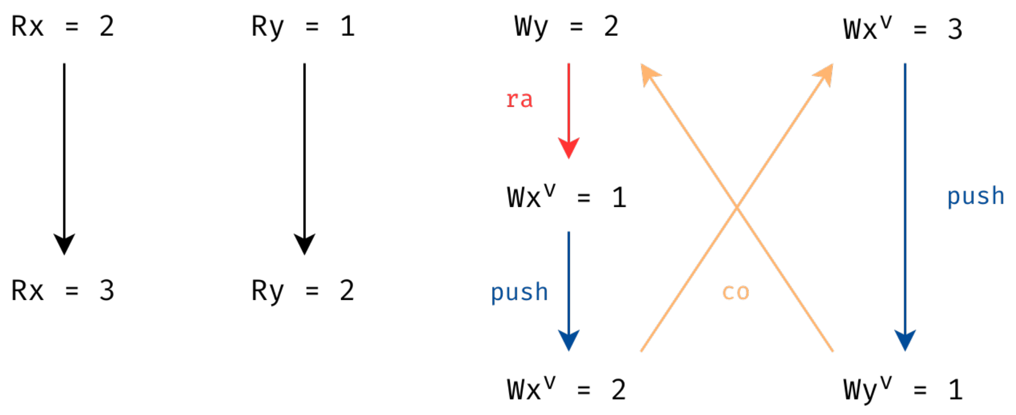


Figure T.3: Execution graph before write-write merge on Volatile (Forbidden)

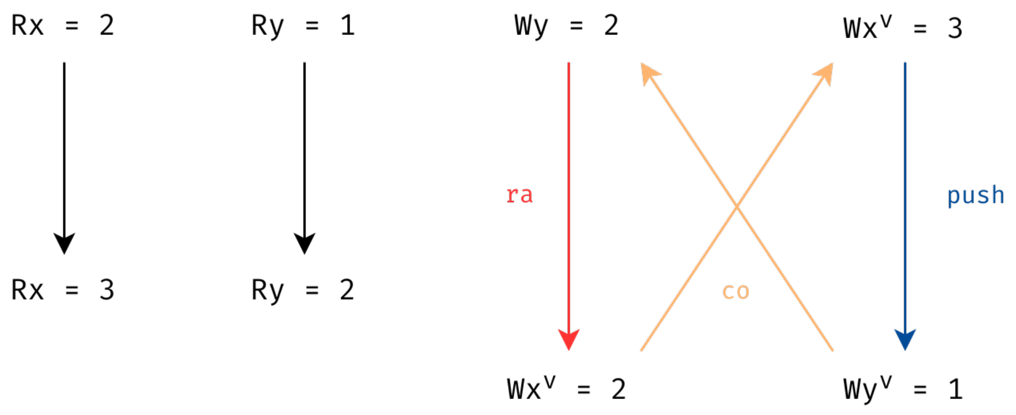


Figure T.4: Execution graph after write-write merge on Volatile (Allowed)

- $G_{src}.\mathtt{po} = G_{tgt}.\mathtt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a,b\rangle\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.AccessMode(b) \sqsubseteq \mathsf{Opaque}$

Proof. We show that G_{src} fulfills the two requirements to be Jam21-consistent.

- No-Thin-Air. First note that, by the well-formedness of $\mathtt{rf}$ order, a is the only access in the execution graph that has a $\mathtt{rf}$ edge to b. Therefore, $a \xrightarrow{\mathtt{rf}} b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$ implies that $a \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$, which means there is also a $(\mathtt{po}|\mathtt{rf})+$ cycle in G_{tgt}, contradicting to our previous assumption.

- Coherence. Since $AccessMode(b) = \mathsf{Opaque}$, there is no out-going cross-thread edge from b and for all event i such that $b \xrightarrow{\mathtt{vo}} i$, we have $a \xrightarrow{\mathtt{vo}} i$ (similarly, for all event j such that $j \xrightarrow{\mathtt{vo}} b$, we have $j \xrightarrow{\mathtt{vo}} a$). Since $a \xrightarrow{\mathtt{rf}} b$ is intra-thread, for any $\mathtt{co}$ edge derived from $a \xrightarrow{\mathtt{rf}} b$ using the `corr` rule, it implies that there is a read access r and write access w such that $a \xrightarrow{\mathtt{rf}} b \xrightarrow{\mathtt{po}} r$ and $w \xrightarrow{\mathtt{rf}} r$ we can derive the same $\mathtt{co}$ edge using the `cowr` rule with $a \xrightarrow{\mathtt{vo}} r$ and $w \xrightarrow{\mathtt{rf}} r$. Similarly, for any $\mathtt{co}$ edge derived from $a \xrightarrow{\mathtt{rf}} b$ using the `cowr` rule, it implies that there is a write access w such that $w \xrightarrow{\mathtt{vo}} b$. Then $w \xrightarrow{\mathtt{vo}} a$ as well. Using the `coww` rule we can derive the same $\mathtt{co}$ edge. Thus, if there is any $\mathtt{co}$ cycle in G_{src}, the same $\mathtt{co}$ cycle also appear in G_{tgt}, contradicting to our previous assumption.

□

Counter Example Here we show that write/RMW-read merging is not valid if the read is or is stronger than Acquire mode. Consider the following example:

```
Thread0 {
int r1 = X.getOpaque(); // 1
int r2 = X.getOpaque(); // 2
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }

Thread2 {
    Y.setOpaque(1);
  }
```

```
Thread3 {
X.setRelease(2);
int r7 = X.getAcquire(); // 2
int r5 = Z.getVolatile(); // 0
int r6 = Y.getVolatile(); // 1
}

Thread4 {
    Y.setVolatile(2);
    X.setVolatile(1);
  }
```

The execution graph can be found in Fig. T.5. The execution is forbidden. Indeed, there are two possible cases:

1. $r_z^V = 0 \xrightarrow{\texttt{vvo}} w_x^V = 1$. Since `rf` $\subseteq$ `vvo` and `ra` $\subseteq$ `vvo`, we can infer that $w_x^{\texttt{rel}} = 2 \xrightarrow{\texttt{vvo}} r_z^V = 0 \xrightarrow{\texttt{vvo}} w_x^V = 1$. Using the `coww` rule, we can infer that $w_x^{\texttt{rel}} = 2 \xrightarrow{\texttt{co}} w_x^V = 1$, which contradicts with the `co` edge we inferred using the `corr` rule and Thread 0.

2. $w_y^V = 2 \xrightarrow{\texttt{vvo}} r_y^V = 1$. This immediately contradicts with the `co` edge we derived using the `corr` rule with Thread 1.

Applying the transformation, we have:

```
Thread0 {
int r1 = X.getOpaque(); // 1
int r2 = X.getOpaque(); // 2
}

Thread1 {
    int r3 = Y.getOpaque(); // 1
    int r4 = Y.getOpaque(); // 2
  }

Thread2 {
    Y.setOpaque(1);
  }
```

```
Thread3 {
X.setRelease(2);
int r7 = 2;
int r5 = Z.getVolatile(); // 0
int r6 = Y.getVolatile(); // 1
}

Thread4 {
    Y.setVolatile(2);
    X.setVolatile(1);
  }
```

The execution graph is shown in Fig. T.6. Due to the removal of the `rf` and `ra` edge, the previously forbidden behavior is introduced after the transformation.

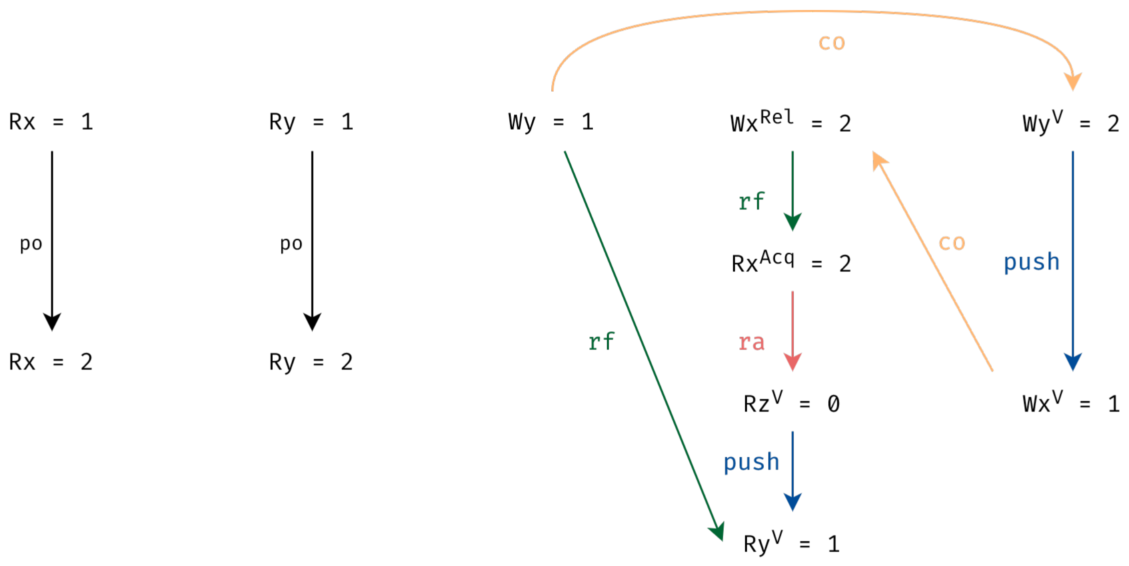


Figure T.5: Execution Graph before Write-read Merge Transformation (Forbidden)

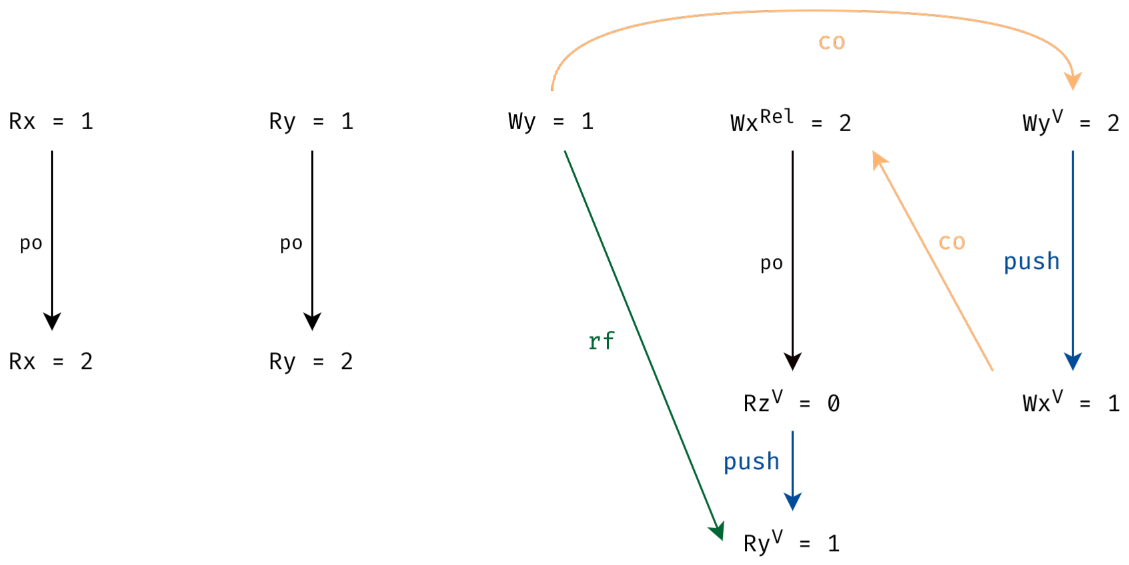


Figure T.6: Execution Graph after Write-read Merge Transformation (Allowed)

T.2.4 Write-RMW Merging

▶ THEOREM 16 (WRITE-RMW MERGING)
Let G_{tgt} be a JAM21-consistent execution. Let $b \in G_{tgt}.\mathsf{Wrts} \setminus G_{tgt}.\mathtt{RMW}$, $a \notin G_{tgt}.\mathsf{Evts}$ and $v \in \mathtt{Val}$. There exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{a\}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.AccessMode(a) \in \{\mathsf{Opaque}, \mathsf{Release}\}$
- $G_{src}.AccessMode(b) \in \{\mathsf{Acquire}, \mathsf{Release}\}$
- $G_{src}.loc(b) = G_{src}.loc(a)$
- $b \in G_{src}.\mathtt{RMW}$
- $G_{src}.val(b) = (G_{src}.val(a), v)$
- $G_{src}.\mathtt{po} = G_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $G_{src}.\mathtt{rf} = G_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $G_{src}.\mathtt{to} = G_{tgt}.\mathtt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{to}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{to}} j\}$
- $G_{src}.\mathtt{IW} = G_{tgt}.\mathtt{IW}$

and G_{src} is JAM21-consistent.

Proof. Most parts of the proof is similar to the proof for write-write merging except for the case where there is a $\mathtt{co}$ cycle in G_{src} due to the total coherence order among RMW operations. Suppose G_{src} violates COHERENCE by having a $\mathtt{co}$ cycle built from the `cormwtotal` rule. That is, we have a RMW operation i such that:

- If $b \xrightarrow{\mathtt{cormwtotal}} i$, then $i \xrightarrow{\mathtt{co}} b$
- If $i \xrightarrow{\mathtt{cormwtotal}} b$, then $b \xrightarrow{\mathtt{co}} i$

Note that we cannot use existing $\mathtt{co}$ orders to derive other orders than $\xrightarrow{\mathtt{cormwexcl}}$ orders (which is also a $\mathtt{co}$ order). If the $\mathtt{co}$ between i and b are not $\xrightarrow{\mathtt{cormwexcl}}$ edges, then they co-exists in one execution. Now we have $i \xrightarrow{\texttt{co}} b \xrightarrow{\texttt{co}} i$. If the co between i and b are $\xrightarrow{\texttt{cormwexcl}}$ edges, then there exist two RMW operations j and k, such that, $b \xrightarrow{\texttt{rf}} j$, $i \xrightarrow{\texttt{rf}} k$, $i \xrightarrow{\texttt{co}} j$ and $b \xrightarrow{\texttt{co}} k$. Note that there is still a total order among i, j, k in G_{tgt}. Now we have either $j \xrightarrow{\texttt{cormwtotal}} k$ or $k \xrightarrow{\texttt{cormwtotal}} j$. Each case yields a contradiction by the `coermwexcl` rule. Therefore, if there is a co cycle in G_{src}, G_{tgt} is also forbidden, which contradicts to our previous assumption. □

T.2.5 RMW-RMW Merging

▶ Theorem 17 (RMW-RMW Merging)
Let G_{tgt} be a Jam21-consistent execution. Let x be a memory location and $a \in G_{tgt}.\mathsf{Evts}$ with $G_{tgt}.val(a) = (v_r, v_w)$, $G_{tgt}.loc(a) = x$, and $G_{tgt}.AccessMode(a) \in \{\mathsf{Release, Acquire}\}$. Let $b \notin G_{tgt}.\mathsf{Evts}$, there exists a G_{src} such that:

- $G_{src}.\mathsf{Evts} = G_{tgt}.\mathsf{Evts} \cup \{b\}$
- $\forall i \in G_{tgt}.\mathsf{Evts}, G_{src}.AccessMode(i) = G_{tgt}.AccessMode(i)$
- $G_{src}.val(a) = (v_r, v)$
- $G_{src}.val(b) = (v, v_w)$
- $G_{src}.loc(b) = x$
- $G_{src}.AccessMode(b) = G_{src}.AccessMode(a) \in \{\mathsf{Release, Acquire}\}$
- $G_{src}.\mathsf{po} = G_{tgt}.\mathsf{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathsf{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathsf{po}} j\}$
- $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf} \cup \{\langle a, b\rangle\}$
- $G_{src}.\texttt{to} = G_{tgt}.\texttt{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\texttt{to}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\texttt{to}} j\}$
- $G_{src}.\texttt{IW} = G_{tgt}.\texttt{IW}$

and G_{src} is Jam21-consistent.

Proof. We show that G_{src} fulfills the two requirements of Jam21-consistency.

- No-Thin-Air. Suppose G_{src} violates this requirement and has a $(\mathsf{po}|\texttt{rf})+$ cycle. Since $G_{src}.val(a) = (v_r, v)$ and $G_{src}.val(b) = (v, v_w)$, if $a \xrightarrow{\mathsf{po}} b \xrightarrow{(\mathsf{po}|\texttt{rf})+} a$ in G_{src}, it implies that $a \xrightarrow{(\mathsf{po}|\texttt{rf})+} a$ in G_{tgt}, contradicting to our previous assumption.

- Coherence. First note that there is only one rf edge from a in G_{src} and that is $a \xrightarrow{\texttt{rf}} b$. In addition, for all event i such that $i \xrightarrow{\texttt{vo}} b$ in G_{src}, $i \xrightarrow{\texttt{vo}} a$ in G_{tgt}. For all j such that $b \xrightarrow{\texttt{vo}} j$ in G_{src}, $a \xrightarrow{\texttt{vo}} j$ in G_{tgt}. Therefore, if there is a co cycle in G_{src}, there is also a co cycle in G_{tgt}, contradicting to our previous assumption.

□

T.3 Register Promotion for non-shared Variable

▶ Theorem 18 (Weakening for non-shared variable)
Let G_{tgt} be a Jam21-consistent execution such that, for all accesses i and j in G_{tgt}.Evts, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in G_{tgt}$.Evts, $loc(i) = x \Rightarrow AccessMode(i) =$ Plain. There exists an execution G_{src} such that:

- G_{src}.Evts = G_{tgt}.Evts
- G_{src}.po = G_{tgt}.po
- G_{src}.rf = G_{tgt}.rf
- G_{src}.to = G_{tgt}.to
- G_{src}.IW = G_{tgt}.IW
- $\forall i \in G_{src}$.Evts, $loc(i) = x \Rightarrow AccessMode(i) \in$ {Release, Acquire}

and G_{src} is Jam21-consistent.

Proof. We show that G_{src} fulfills the two requirements of Jam21-consistency.

1. No-Thin-Air. Note that there is no cross-thread rf edge from or to accesses of location x. Therefore, since G_{src}.po = G_{tgt}.po and G_{src}.rf = G_{tgt}.rf, if there is a $(\texttt{po}|\texttt{rf})+$ cycle in G_{src}, there is a $(\texttt{po}|\texttt{rf})+$ cycle in G_{tgt}, contradicting to our previous assumption.

2. Coherence. Note that the transformation is equivalent to removing all the ra edges that involve accesses to x. Therefore, $G_{src}.\texttt{vo} = G_{tgt}.\texttt{vo} \setminus \{\langle a, b\rangle \mid (loc(a) = x \wedge a \xrightarrow{\texttt{ra}} b \wedge AccessMode(b) \neq \texttt{Release}) \vee (loc(b) = x \wedge a \xrightarrow{\texttt{ra}} b \wedge AccessMode(a) \neq \texttt{Acquire})\}$.

For accesses i and j such that $loc(i) \neq x$ and $loc(j) \neq x$, if $i \xrightarrow{\texttt{vo}} j$ in G_{src}, $i \xrightarrow{\texttt{vo}} j$ in G_{tgt}. In addition, since x is not shared across different threads, all accesses to location x are related by po. Since all accesses to x have an access mode of either Release or Acquire, there is no cross-thread `vo` edges or `rf` edges from or to these accesses. Therefore, for all memory location $y \neq x$, $G_{src}.\texttt{vo} \restriction_y = G_{tgt}.\texttt{vo} \restriction_y$. Suppose G_{src} violates this requirement by having a `co` cycle:

- If there is a `co` cycle with accesses to location x. Since $G_{src}.\texttt{po-loc} = G_{tgt}.\texttt{po-loc}$ and $\texttt{po-loc} \subseteq \texttt{vo}$, then there is also a `co` cycle with accesses to location x in G_{tgt}, contradicting to our previous assumption.
- If there is a `co` cycle with accesses to other locations. Since for all memory location $y \neq x$, $G_{src}.\texttt{vo} \restriction_y = G_{tgt}.\texttt{vo} \restriction_y$ and $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf}$, it implies there is also a `co` cycle in G_{tgt}, contradicting to our previous assumption.

□

▶ THEOREM 19 (REMOVING PLAIN ACCESSES FOR NON-SHARED VARIABLE)
Let G_{tgt} be a JAM21-consistent execution. Let x be a memory location and for all $i \in G_{tgt}.\text{Evts}$ such that $loc(i) = x, Tid(i) = t$ for some x and t. Let $a \notin G_{tgt}.\text{Evts}$. There is a G_{src} such that:

- $G_{src}.\text{Evts} = G_{tgt}.\text{Evts} \cup \{a\}$
- $G_{src}.loc(a) = x$
- $G_{src}.AccessMode(a) = \text{Plain}$
- $G_{src}.\text{po} \supset G_{tgt}.\text{po}$
- for all $i \in G_{src}.\text{Evts}$ such that $G_{src}.loc(i) = x$, $i \xrightarrow{\text{po}} a$ or $a \xrightarrow{\text{po}} i$
- $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf}$ if $a \in G_{src}.\text{Wrts} \backslash \texttt{RMW}$, otherwise, $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf} \cup \{\langle i, a \rangle\}$ such that $(i \in G_{src}.\text{Wrts}) \wedge (loc(i) = x) \wedge (i \xrightarrow{\text{po}} a) \wedge (\forall j \in G_{src}.\text{Evts}, (loc(j) = x) \wedge (j \xrightarrow{\text{po}} a) \Rightarrow (j \xrightarrow{\text{po}} i))$.
- $G_{src}.\texttt{to} = G_{tgt}.\texttt{to}$
- $G_{src}.\texttt{IW} = G_{tgt}.\texttt{IW}$

and G_{src} is JAM21-consistent.

Proof. It is clear that G_{src} does not violate NO-THIN-AIR and there is no co cycle for accesses to location x. For COHERENCE, note that for all memory location $y \neq x$, $G_{src}.\texttt{vo} \restriction_y = G_{tgt}.\texttt{vo} \restriction_y$ and $G_{src}.\texttt{rf} = G_{tgt}.\texttt{rf}$, it implies that if there is a co cycle in G_{src} there is also a co cycle in G_{tgt}, contradicting to our previous assumption. □

[ADG21] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. "Armed Cats." *ACM Transactions on Programming Languages and Systems*, **43**:1–54, 6 2021.

[AFI09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. "The semantics of power and ARM multiprocessor machine code." *Proceedings of the 4th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP'09*, pp. 13–24, 2009.

[AH90] Sarita V. Adve and Mark D. Hill. "Weak ordering—a new definition." *ACM SIGARCH Computer Architecture News*, **18**:2–14, 6 1990.

[AMS12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. "Fences in weak memory models (extended version)." *Formal Methods in System Design*, **40**:170–205, 4 2012.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory." *ACM Trans. Program. Lang. Syst.*, **36**, 7 2014.

[BMO12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. "Clarifying and compiling C/C++ concurrency." *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 509–520, 1 2012.

[BP19] John Bender and Jens Palsberg. "A formalization of Java's concurrent access modes." *Proceedings of the ACM on Programming Languages*, **3**:1–28, 10 2019.

[Cor03] International Business Machines Corporation. *PowerPC User Instruction Set Architecture*, 2003.

[Cor08] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols)*, 2008.

[CYW21] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. "Sound and efficient concurrency bug prediction." *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, **21**:255–267, 8 2021.

[Dev07] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual (3 vols)*, 2007.

[FF09] Cormac Flanagan and Stephen N. Freund. "FastTrack: efficient and precise dynamic race detection." *SIGPLAN Not.*, **44**(6):121–133, jun 2009.

[GRX19] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. "Dependence-Aware, Unbounded Sound Predictive Race Detection." *Proc. ACM Program. Lang.*, **3**(OOPSLA), oct 2019.

[HH16] Shiyou Huang and Jeff Huang. "Maximal Causality Reduction for TSO and PSO." *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 447–461, October 2016.

[HLR15] Jeff Huang, Qingzhou Luo, and Grigore Rosu. "GPredict: Generic Predictive Concurrency Analysis." *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, **1**:847–857, 5 2015.

[HMR14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. "Maximal sound predictive race detection with control flow abstraction." *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, **49**:337–348, 6 2014.

[ISO98] ISO. *ISO/IEC 14882:1998: Programming languages — C++*, September 1998. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[KLV23] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. "Kater: Automating Weak Memory Model Metatheory and Consistency Checking." *Proc. ACM Program. Lang.*, **7**(POPL), jan 2023.

[KMV17] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. "Dynamic race prediction in linear time." *ACM SIGPLAN Notices*, **52**:157–170, 6 2017.

[KP18] Christian Gram Kalhauge and Jens Palsberg. "Sound deadlock prediction." *Proceedings of the ACM on Programming Languages*, **2**, 11 2018.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system." *Commun. ACM*, **21**(7):558–565, jul 1978.

[Lam79] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Transactions on Computers*, **C-28**:690–691, 9 1979.

[LBP22] Shuyang Liu, John Bender, and Jens Palsberg. "Compiling Volatile Correctly in Java." In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, 2022.

[LCK23] Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. "Fair Operational Semantics." *Proc. ACM Program. Lang.*, **7**(PLDI), jun 2023.

[Lea11a] Doug Lea. "The JSR-133 Cookbook for Compiler Writers.", 2011. Last modified: Tue Mar 22 07:11:36 2011.

[Lea11b] Doug Lea. "The JSR-133 Cookbook for Compiler Writers.", 2011. Last modified: Tue Mar 22 07:11:36 2011.

[Lea18] Doug Lea. "Using JDK 9 Memory Order Modes.", 2018. Last Updated: Fri Nov 16 08:46:48 2018.

[LNO20] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. "Making Weak Memory Models Fair." *arXiv preprint arXiv:2012.01067*, 12 2020.

[LV16] Ori Lahav and Viktor Vafeiadis. "Explaining Relaxed Memory Models with Program Transformations." *FM 2016: Formal Methods*, pp. 479–495, 2016.

[LVK17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing Sequential Consistency in C/C++11." *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 618–632, 2017.

[MKV18] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. "What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection." *Proceedings of the ACM on Programming Languages*, **2**:1–29, 10 2018.

[ML21] Roy Margalit and Ori Lahav. "Verifying observational robustness against a c11-style memory model." *Proceedings of the ACM on Programming Languages*, **5**:1–33, 1 2021.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java Memory Model." *ACM SIGPLAN Notices*, **40**:378–391, 2005.

[MPV21] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. "Optimal prediction of synchronization-preserving races." *Proceedings of the ACM on Programming Languages*, **5**, 1 2021.

[MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models.", 2012. Draft available from http://www. cl. cam. ac. uk/ pes20/ppc-supplemental/test7.pdf.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO." *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, p. 391–407, 2009.

[Pav20] Andreas Pavlogiannis. "Fast, Sound, and effectively complete dynamic race prediction." *Proceedings of the ACM on Programming Languages*, **4**:1–29, 1 2020.

[PFD18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8." *Proceedings of the ACM on Programming Languages*, **2**:1–29, 1 2018.

[PLV19] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. "Bridging the gap between programming languages and hardware weak memory models." *Proceedings of the ACM on Programming Languages*, **3**, 1 2019.

[SCR13] Traian Florin ŞerbănuŢă, Feng Chen, and Grigore Roşu. "Maximal Causal Models for Sequentially Consistent Systems." *Runtime Verification*, pp. 136–150, 2013.

[SES12] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. "Sound Predictive Race Detection in Polynomial Time." *SIGPLAN Not.*, **47**(1):387–400, jan 2012.

[Shi21] Aleksey Shipilev. "[JDK-8262877] PPC sequential consistency problem: volatile stores are too weak." Technical report, OpenJDK Bug System, 03 2021.

[SRA05] Koushik Sen, Grigore Roşu, and Gul Agha. "Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions." *Formal Methods for Open Object-Based Distributed Systems*, pp. 211–226, 2005.

[SSA11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER multiprocessors." *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, p. 175, 2011.

[SSO10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. "X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors." *Communications of the ACM*, **53**:89–97, 7 2010.

[SWY11] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. "Generating Data Race Witnesses by an SMT-based Analysis." *NASA Formal Methods Symposium*, 2011.

[TAC23] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. "Optimal Reads-From Consistency Checking for C11-Style Memory Models." *Proc. ACM Program. Lang.*, **7**(PLDI), jun 2023.

[TMP23] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. "Sound Dynamic Deadlock Prediction in Linear Time." *Proceedings of the ACM on Programming Languages*, **7**:1733–1758, 6 2023.

[VBC15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. "Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it." *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, **50**:209–220, 1 2015.

[WPP20] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu Yu Guo. "Repairing and mechanising the JavaScript relaxed memory model." *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 346–361, 6 2020.

[Še08] Jaroslav Ševčík. *Program Transformations in Weak Memory Models.* PhD thesis, The University of Edinburgh, 2008. Publication Title: Memory.