Correct Compilation of Relaxed Memory Concurrency

Thesis approved by the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Soham Sundar Chakraborty

Date of Defense: 16.07.2019

Dean: Prof.Dr. Klaus Schneider

Reviewer: Dr. Mark Batty

Reviewer: Prof.Dr. Rupak Majumdar

Reviewer: Dr. Viktor Vafeiadis

D 386

Summary

Shared memory concurrency is the pervasive programming model for multicore architectures such as x86, Power, and ARM. Depending on the memory organization, each architecture follows a somewhat different shared memory model. All these models, however, have one common feature: they allow certain outcomes for concurrent programs that cannot be explained by interleaving execution. In addition to the complexity due to architectures, compilers like GCC and LLVM perform various program transformations, which also affect the outcomes of concurrent programs.

To be able to program these systems correctly and effectively, it is important to define a formal language-level concurrency model. For efficiency, it is important that the model is weak enough to allow various compiler optimizations on shared memory accesses as well as efficient mappings to the architectures. For programmability, the model should be strong enough to disallow bogus "out-of-thin-air" executions and provide strong guarantees for well-synchronized programs. Because of these conflicting requirements, defining such a formal model is very difficult. This is why, despite years of research, major programming languages such as C/C++ and Java do not yet have completely adequate formal models defining their concurrency semantics.

In this thesis, we address this challenge and develop a formal concurrency model that is very good both in terms of compilation efficiency and of programmability. Unlike most previous approaches, which were defined either operationally or axiomatically on single executions, our formal model is based on event structures, which represents multiple program executions, and thus gives us more structure to define the semantics of concurrency.

In more detail, our formalization has two variants: the weaker version, WEAKEST, and the stronger version, WEAKESTMO. The WEAKEST model simulates the promising semantics proposed by Kang et al., while WEAKESTMO is incomparable to the promising semantics. Moreover, WEAKESTMO discards certain questionable behaviors allowed by the promising semantics. We show that the proposed WEAKESTMO model resolve out-of-thin-air problem, provide standard data-race-freedom (DRF) guarantees, allow the desirable optimizations, and can be mapped to the architectures like x86, PowerPC, and ARMv7. Additionally, our models are flexible enough to leverage existing results from the literature to establish data-race-freedom (DRF) guarantees and correctness of compilation.

In addition, in order to ensure the correctness of compilation by a major compiler, we developed a translation validator targeting LLVM's "opt" transformations of concurrent C/C++ programs. Using the validator, we identified a few subtle compilation bugs, which were reported and were fixed. Additionally, we observe that LLVM concurrency semantics differs from that of C11; there are transformations which are justified in C11 but not in LLVM and vice versa. Considering the subtle aspects of LLVM concurrency, we formalized a fragment of LLVM's concurrency semantics and integrated it into our WEAKESTMO model.

Kurzfassung

Shared Memory Concurrency ist das gängige Programmiermodell für Multicore-Architekturen wie x86, Power oder ARM. Je nach Speicherorganisation nutzt jede Architektur ein etwas anderes Shared-Memory-Modell. All diese Modelle haben jedoch ein gemeinsames Merkmal: Sie erlauben bestimme Verhalten gleichzeitiger Programme, die nicht durch Interleaving-Ausführung erklärt werden können. Neben der architekturbedingten Komplexität wirken sich auch Programmtransformationen, wie sie von Compilern wie GCC und LLVM ausgeführt werden, auf das Verhalten nebenläufigern Programme aus.

Um solche Systeme korrekt und effektiv zu programmieren, muss ein formales Parallelitätsmodell auf Sprachenebene definiert werden. Aus Effizienzgründen ist es wichtig, dass das Modell schwach genug ist, um verschiedene Compiler-Optimierungen bei Shared-Memory-Zugriffen sowie effiziente Abbildung auf verschiedene Architekturen zu ermöglichen. Aus Gründen der Programmierbarkeit sollte das Modell stark genug sein, um Öut-of-Thin-AirAusführungen zu verhindern und starke Garantien für gut synchronisierte Programme zu bieten. Diese gegensetzlichen Anforderungen machen die Definition eines solchen formalen Modells sehr schwierig. Auf diesem Grund verfügen die wichtigsten Programmiersprachen wie C/C++ und Java trotz jahrelanger Forschung noch nicht über vollständig geeignete formale Modelle ihrer Parallelitätssemantik.

In dieser Arbeit stellen wir uns dieser Herausforderung und entwickeln ein formales Parallelitätsmodell, das sowohl hinsichtlich der Kompilierungseffizienz als auch der Programmierbarkeit sehr gut ist. Im Gegensatz zu den meisten früheren Ansätzen, die entweder operational oder axiomatisch auf Basis einzelner Ausführungen definiert werden, basiert unser formales Modell auf Event-Strukturen, die mehrere Programmausführungen gleichzeitig abbilden, und gibt uns somit mehr Struktur, um die Semantik der Parallelität zu definieren.

Im Detail hat unserere Formalisierung zwei Varianten: die schwächere Version WEAKEST und die stärkere Version WEAKESTMO. Das WEAKEST Modell simuliert die “Promising Semantics”, die von Kang et al. vorgeschlagen wurde, wohingegen das WEAKESTMO Modell mit den “Promising Semantics” nicht vergleichbar ist. Darüber hinaus verhindert WEAKESTMO bestimmte problematische Verhaltensweisen, die die “Promising Semantics” zulassen. Wir zeigen, dass die vorgeschlagenen WEAKESTMO Modell das Out-of-Thin-Air-Problem lösen, übliche DRF-Garantien (Data Race Freedom) einhalten, die gewünschten Optimierungen zulassen und auf die Architekturen wie x86, PowerPC, und ARMv7 abgebildet werden können. Darüber hinaus sind unsere Modelle flexibel genug, um vorhandene Ergebnisse aus der Literatur nutzbar zu machen.

Weiterhin haben wir einen Translation Validator entwickelt, um die Korrektheit der Kompilierung durch einen weitverbreiteten Compiler sicherzustellen, indem wir die LLVM-“opt”-Transformationen von parallelen C/C++-Programmen prüfen. Mit Hilfe des Validators haben wir einige subtile Kompilierungsfehler identifiziert, die gemeldet und behoben wurden. Au-

ßerdem stellen wir fest, dass die LLVM-Parallelitätssemantik sich von der von C11 unterscheidet: Es gibt Transformationen, die in C11 gerechtfertigt sind, nicht aber in LLVM, und umgekehrt. In Anbetracht dieser subtilen Aspekte der LLVM-Parallelität haben wir ein Fragment der LLVM-Parallelitätssemantik formalisiert und in unser WEAKESTMO Modell integriert.

Acknowledgments

The PhD journey has been a truly life-changing experience for me. This journey would not have been possible without the guidance and support from a great number of individuals.

First and foremost, I would like to extend my sincere gratitude to my adviser Dr. Viktor Vafeiadis for his dedicated guidance, help, encouragement and continuous support throughout my PhD. I consider myself extremely lucky to have him as my supervisor who cared so much about my work. He introduced me to this exciting field of relaxed memory concurrency and formal methods. His knowledge is truly awe-inspiring and I have learnt so much from him. He has also taught me extensively the of value systematic thinking and has shown me the way to pursue high quality research. His dedication to research and work practices have made a deep impression on me and would be invaluable in my future endeavors.

Besides my adviser, I would like to sincerely thank my committee members, Prof. Rupak Majumdar and Dr. Mark Batty for their time, advice, and valuable feedback on my research. Additionally, I also got many opportunities to interact with other MPI-SWS faculties and I am deeply thankful to them for the enriching discussions and many valuable suggestions. I would also like to thank MPI-SWS institute for the generous funding, administrative support, and for providing a vibrant research environment.

A number of peers have helped me to shape my research. I would like to sincerely thank Marko Doko, Ori Lahav, Evgenii Moiseenko, Anton Podkopaev, Azalea Raad, and anonymous reviewers for insightful discussions and valuable feedback in various aspects of my work.

Next, I would like to thank my friends at MPI-SWS including (alphabetically) Azalea, Burcu, Ivan, Johannes, Kaushik, Mainack, Marko, Michalis, Ori, Simin, Utkarsh, and everybody that I have forgotten to mention for various discussions and suggestions.

I am deeply indebted to a number of people around who have generously helped me and my family to survive in a foreign land. Among them Vera Schreiber, Marko Doko, Johannes Kloos were the ‘go-to’ persons for many day-to-day life issues. I am extremely thankful for all their unconditional help and support.

I extend my gratitude to my parents, Swapna Chakraborty and Shyamsundar Chakraborty, sister, Soma Chakraborty, for their endless love, prayers, and constant encouragement. I sincerely thank my in-laws, Deepali Chatterjee and Promothesh Chatterjee, for their love and support. I also thank my uncle, Raghunath Banerjee, who inspired me to pursue PhD.

Finally, I thank my wife, Nivedita and my son, Proneel, who have accompanied me during my PhD. Without their love, care, understanding, and sacrifices this journey would not have been possible. I dedicate my thesis to them.

Contents

1 Introduction

The advent of multicore era has revolutionized the high performance computing landscape. In multicore systems, instead of relying solely on the performance of a single processing unit, higher performance is achieved by concurrent execution on multiple cores. Cores are connected to main memory and the main memory is shared among all the cores.

Shared memory concurrency emerged as a pervasive programming paradigm to exploit multicore systems. In this paradigm, multiple threads run on multiple cores and the shared data reside at the main memory.

To reason about such programs, we need to consider the outcomes of all its possible executions, which we term as program behavior.

The program behaviors of a given program are described by the underlying memory consistency model. The simplest such model is *sequential consistency* (SC), which can be explained in terms of interleaving: "*... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*" [38].

For concreteness, consider the following program:

$$\begin{array}{l|l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

In this program, X and Y are shared variables initialized to zeros and a and b are local variables. According to SC, there are three possible program behaviors: (1) $a = b = 0$, (2) $a = 0 \wedge b = 1$, and (3) $a = 1, b = 0$. In contrast, note that $a = b = 0$ is not allowed by SC, as no interleaving of the threads of the programs yields that outcome. Essentially, SC explains the program behaviors of a system whose memory accesses directly affect the shared memory.

Maintaining the ordering constraints of SC in practice, however, is often prohibitively expensive in terms of performance. As a result, multicore systems use caching scheme that violate SC. Let's return to the example program in the presence of caches, where we assume that each thread runs on a different core, each having its own private cache. The program can execute as follows. First, the writes on X and Y go to the caches (the write of X goes to the cache of the first core, while the write of Y to that of the second core). Next, the reads of X and Y result in cache misses in their respective threads and fetch the values from the main memory. Finally the values of X and Y reach the main memory. The program can therefore return the non-SC behavior $a = b = 0$.

To allow such behaviors, the architectures follow non-sequentially consistent models based on the underlying organizations of the respective systems. These models are referred to as relaxed/weak memory models.

Apart from architectural optimizations, the additional benefit of relaxed memory models is the flexibility of program transformations on shared memory accesses. Going back to the

example above, a memory model which allows the $a = b = 0$ program behavior in turn allows the reordering of the access pairs in each thread of this program. These reorderings followed by an interleaving execution where the reads of X and Y take place before the writes also explains the $a = b = 0$ program behavior. Note that even without any caching effect from the architecture, some of the relaxed memory behaviors are observable due to certain program transformations (e.g. reordering). These transformations on shared memory accesses enable additional optimizations and in turn gain further performance improvement.

To incur the above mentioned benefits, similar to the architectures, modern programming languages also have relaxed memory models. The relaxed consistency models of the architectures influence the memory models of the programming languages such as C/C++, Java and so on. These programming languages provide architecture independent programming abstractions, allowing a programmer to specify the required consistency level for shared memory accesses. Among different memory models of the programming languages, we study the memory model of C/C++ which is introduced in the 2011 C/C++ standards (ISO/IEC 9899:2011 [30]; ISO/IEC 14882:2011 [29]), henceforth C11.

C11 introduced atomic variables and memory fences and a set of memory orders for the atomic accesses and fences. In addition, C11 specifies a set of memory consistency constraints for the shared memory accesses and fences. These constructs provide a higher-level platform-independent abstraction over the concurrency semantics of existing multi-core hardware implementations. Programmers use these constructs to write programs with required consistency guarantees and these C11 constructs are mapped to the instructions in the target multicore hardware in optimal fashion. The C11 model, similar to the architectures, also intends to allow various shared memory transformations (e.g. reordering, elimination etc) based on the access types and memory orders.

A number of state-of-the-art optimizing compilers like GCC and LLVM support the C11 compilation. These compilers do not directly map C11 concurrency constructs to the architectures, instead, the compilation involves multiple optimizing transformation steps. First, a C11 program is is mapped to the compiler's intermediate representation, then a number of optimizations are performed, and finally the target machine code is generated. To perform the transformations, the compilers also follow certain concurrency semantics. Thus the compilation of relaxed memory concurrent program involves the interplay between multiple relaxed memory models at the programming language, compiler, and architecture levels.

Considering the complexities involved, ensuring correctness of C11 compilation is a non-trivial task which constitutes the main motivation of this thesis. To address this issue the first step is to identify the correct transformations for C11 relaxed memory concurrency and the next step is to check whether a compiler performs only these correct transformations. Each of these steps involves a number technical challenges which we discuss in more details.

- *Correctness of Transformations.* A transformation is correct when the target, the program after the transformation, does not generate any new outcomes compared to the source, the program before the transformation. At the same time the set of valid outcomes of a given program is decided by the the programming language semantics. Hence the correctness of a transformation depends on the semantics of the source and the target programming languages. In this scenario a relaxed semantics enables many desirable transformations in

case of concurrency.

Apart from the transformation correctness, a semantics should also ensure certain guarantees for programmability purpose. To provide programmability, a semantics should have strong enough constraints to ensure *data-race-freedom* (DRF) guarantees so that the semantics does not justify an weaker outcome with data race for a well-synchronized program. Additionally, the semantics should disallow some program outcome which does not occur in any actual execution. Such behavior is termed as 'out-of-thin-air' (OOTA) behaviors in the context of relaxed memory concurrency. The DRF guarantees and eliminating OOTA behaviors are crucial in reasoning about program behaviors.

Thus a concurrency semantics requires to serve conflicting requirements: the semantics must be strong enough to ensure DRF guarantees as well as to eliminate OOTA behaviors and weak enough to allow desirable transformations.

- *Compiler Correctness.* Once the set of correct transformations are identified, a compiler must apply only the correct transformations in order to preserve the correctness of the entire transformations. A verified compiler provides proofs to ensure the correctness of its transformations and other compilers requires testing or translation validation techniques to ensure the correctness compilation.

There are a number of efforts to address the above discussed technical challenges. The complexities and subtle issues involved in relaxed memory concurrency semantics and concurrency compilation require formal models to reason about the correctness. Batty et al. [11] was the first to formalize the concurrency model from the C++ standard. Sadly, that model was problematic in that it admitted *out-of-thin-air* behavior [24]. Since then, various alternative approaches have tried to find a 'sweet-spot' model to address all the concerns.

Compiler correctness for relaxed memory concurrency is a long-standing research problem which is addressed by various approaches to some extent. For instance, Sevcík et al. [67] is a verified compiler for compilation to TSO memory model. However, so far there exists no verified compiler for C11 relaxed memory concurrency. Considering the difficulty of developing a verified compiler from scratch, an alternative approach is to apply testing and/or validation techniques for the existing C11 compilers to capture compilation bugs. For instance, compiler testing approach proposed by Morisset et al. [48] revealed multiple C11 concurrency compilation bugs in the GCC compiler. Similar to GCC, LLVM also supports C11 concurrency compilation but lacked support for testing and/or validating C11 concurrency compilation.

To address the above discussed challenges, in this thesis we propose a formal model for C11 concurrency semantics which disallows certain OOTA behaviors, ensures DRF guarantees, allows desired compiler optimizations and mappings to target architectures like, x86, PowerPC, and ARMv7. We also establish a connection between our formal models and the promising semantics proposed by Kang et al. [33]. In addition we apply the compiler correctness results to develop a translator validator for 'opt' transformations in the LLVM compiler to check the correctness of the transformations.

Thesis Structure

The thesis is structured as follows.

Background In Chapter 2 we discuss the background details of relaxed memory concurrency and concurrency compilation. We discuss the memory consistency models in various architectures and C11, correctness of transformations in relaxed memory concurrency, and various formal approaches to define relaxed memory models. These semantic models reason about individual executions and suffer from certain limitations to eliminate OOTA behaviors. This limitation is addressed by event structures which facilitate analysis on multiple executions.

The WEAKEST Memory Model In Chapter 3 we start with our proposed formalization WEAKEST (WEAK Event STructure) and show that our proposed scheme eliminates certain OOTA behavior. We define the WEAKEST event structure, its construction, consistency, and the extraction of consistent executions from an WEAKEST event structure.

The WEAKEST (WEAK Event STructure) Model and Promising Semantics In Chapter 4, we show a connection between the proposed WEAKEST model and the 'promising semantics' of Kang et al. [33]. The promising semantics was the first major result resolving the *out-of-thin-air* problem, providing data-race-freedom guarantees, and allowing desired optimizations and mappings to x86, PowerPC, ARMv7 [58] architectures. We show that the proposed WEAKEST model is weaker than promising semantics by a simulation relation.

The WEAKESTMO (WEAK Event STructure with Modification Order) Memory Model In Chapter 5 we define WEAKESTMO which is an extension of WEAKEST model with modification order (mo) at the event structure level. We also demonstrate that WEAKESTMO model is incomparable to the promising semantics [33]; there are certain program behavior which is allowed in WEAKESTMO but not in promising semantics and there are some program behavior which is allowed in promising semantics but not in WEAKESTMO model. Finally, we extend the WEAKESTMO model for C11 and LLVM semantics for the racy programs.

Programmability Results In Chapter 6, we discuss various results concerning the programmability of the proposed memory models. We benchmark the proposed models on the Java causality test cases [45], and show that WEAKESTMO guarantees the standard data-race-freedom (DRF) properties.

Compilation Results In Chapter 7, we establish some results concerning the compilation of WEAKESTMO.We prove the correctness of a number of compiler optimizations as source-to-source transformations as well as the correctness of mappings from WEAKESTMO to various architectures such as x86, PowerPC, ARMv7. (The corresponding results for WEAKEST hold trivially via the promising semantics and the result of Chapter 4.)

Validating LLVM Optimizations In Chapter 8 we describe a translation validator which we develop Based on the earlier results to check the correctness of the transformations in the LLVM 'opt' phase. Using the validator, we identified a few concurrency compilation bugs in LLVM, which we reported and which have since been fixed.

Related Work Chapter 9 describes some further related approaches to define relaxed memory semantics and compares them to WEAKEST and WEAKESTMO. We also discuss the techniques for ensuring compilation correctness in compilers while compiling relaxed memory concurrent programs.

Conclusion We conclude in Chapter 10 with possible future research directions.

Dissemination of this Thesis

The contributions of this thesis are disseminated in following publications

- Grounding Thin-Air Reads with Event Structures.
  Soham Chakraborty, Viktor Vafeiadis.
  In POPL 2019.

- Formalizing the Concurrency Semantics of an LLVM Fragment.
  Soham Chakraborty, Viktor Vafeiadis.
  In CGO 2017.

- Validating Optimizations of Concurrent C/C++ Programs.
  Soham Chakraborty, Viktor Vafeiadis.
  In CGO 2016.

In addition, during my PhD, I have worked on and coauthored the following papers

- Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it.
  Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, Francesco Zappa Nardelli.
  In POPL 2015.

- Aspect-Oriented Linearizability Proofs.
  Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, Viktor Vafeiadis.
  In Logical Methods in Computer Science 11(1) 2015.

2 Background

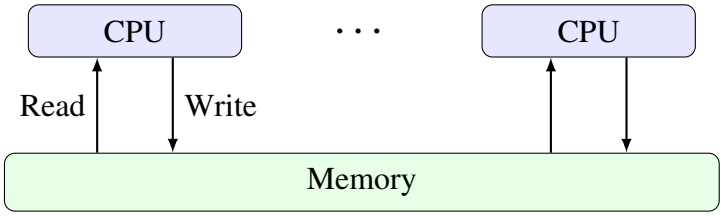
In this chapter, we discuss the relaxed memory models of architectures, programming languages, and compilers. We also discuss various reasoning approaches to justify relaxed memory behaviors. We conclude the chapter with the long-standing challenge in defining a formal semantics of C11 concurrency. Before discussing these details, for concreteness, we start with the syntax of a simple concurrent imperative language.

Notation for Programs The thesis contains a number of small example programs, often called *litmus tests*, that illustrate the various weak memory models. In these examples, we use uppercase letters (e.g., X, Y, etc.) to denote shared variables and lowercase letters (e.g., a, b, etc.) to denote thread-local variables, and write v, v' and so on to range over values manipulated by the program (e.g., integers).

Programs consist of a sequence of initialization writes followed by a fixed parallel composition of a number of threads. Unless otherwise stated, we assume all locations are initialized to zero. We write assignments using C-like syntax. For example, $a = X$ loads the value of shared variable X into the local variable a, while $X = v$ stores the value v into the shared variable X. Finally, to refer to a particular program outcome, we write in appropriately placed comments (// v) the values that certain loads are meant to return.

2.1 Hardware Memory Models

Shared memory concurrency naturally arises in multicore architectures where there are multiple processing units all accessing the same main memory. The organization of the processing units, interconnects, and main memory decides the underlying memory model. We start with a simplest model, known as *sequential consistency* (SC), where the processing units directly access the shared main memory.

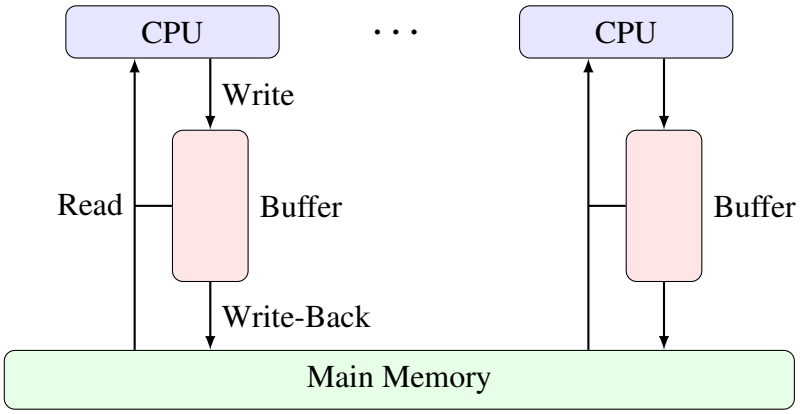


This organization naturally yields a thread interleaving semantics: processors take turns in accessing the shared memory, and the effects of write by a processor are immediately visible to all other processors.

The state-of-the-art architectures, however, do not follow this simple model for performance reasons. They instead have a more complex memory layout with a hierarchy of cache memories and buffers for communicating across computing units. These more complicated memory layouts affect the semantics of programs in subtle ways, which we will now describe.

2.1.1 x86

The x86 architecture introduces a write buffer for each processor as depicted below.



Instructions are issued in-order at each processor. When a write operation takes place, the processor enqueues the write to the respective thread-local write buffer. Initially, this write is observable only to reads from the same thread. At some later point, a *write-back* step propagates the oldest write of the buffer to the memory, which makes the write visible to all other threads. In case of a read operation, the processor reads from the most recent corresponding write in its thread-local write buffer. If no such write is present in the write buffer, the processor reads from the shared main memory.

To illustrate this model, consider the "store buffering" litmus test below.

$$\begin{array}{l|l} X = 1; & Y = 1; \\ a = Y; \quad \text{// } 0 & b = X; \quad \text{// } 0 \end{array} \tag{SB}$$

The annotated outcome $a = b = 0$ is allowed by the following execution. First, the write operation to X executes writing $X = 1$ to the first thread's write buffer. Next, $Y = 1$ is executed, which similarly adds an entry to the second thread's write buffer. Then, the read operations of Y and X get the values from the main memory because the threads do not have a respective store in their local buffers. Hence both reads return 0 and result in $a = b = 0$. Finally, the stores of X and Y are propagated to the main memory.

The concurrency semantics of x86 was first formalized by Sarkar et al. [62], who proposed the x86-CC model. This model, however, allows more outcomes than could be observed on x86 implementations, and was later abandoned in favor of the x86-TSO model by Owens [53], which we described above.

The difference between the two models is illustrated by the outcome $a = 1, b = 0, c = 1, d = 0$ of the "independent reads of independent writes" litmus test below. $a = 1, b = 0, c =$

$1, d = 0$ outcome in the IRIW program below.

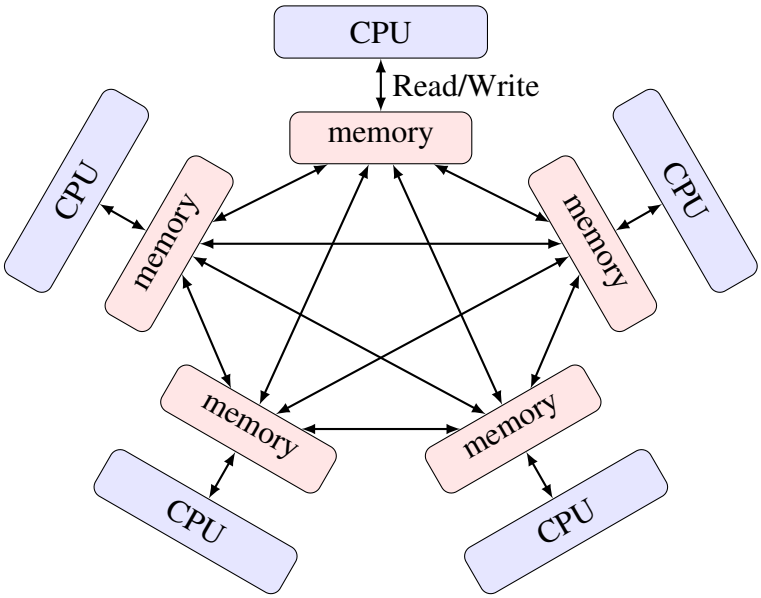
$$X = 1; \left\| \begin{array}{ll} a = X; & // 1 \\ b = Y; & // 0 \end{array} \right\| \left\| \begin{array}{ll} c = Y; & // 1 \\ d = X; & // 0 \end{array} \right\| Y = 1; \qquad \text{(IRIW)}$$

x86-CC model permits the outcome, thereby allowing the stores of X and Y to be observed in different orders by the two middle threads. In contrast, x86-TSO forbids the weak outcome because of the total order in which the stores of X and Y are written back to main memory. If $X = 1$ is written back first, then the third thread cannot return $c = 1 \wedge d = 0$. If, conversely, $Y = 1$ is written back first, then the second thread cannot read $a = 1 \wedge b = 0$.

Owens [53] formalized this *total store order* semantics and called it the x86-TSO model. In fact, they developed two models (an operational one and an axiomatic one) and proved the equivalence between the two models.

2.1.2 PowerPC

The PowerPC architecture is significantly more complex than x86 and exhibits many more relaxed memory behaviors. Along with caches and buffers for each processor, PowerPC also allows pointwise communication between different processors as depicted below.



There have been several attempts to formalize the PowerPC consistency model [63, 47, 12, 64, 8, 35]. Without going into formal details, here we discuss certain aspects of the PowerPC concurrency.

Out-of-Order Execution In PowerPC, the instructions can be executed out of order. To execute a new instruction S, the processor must have already executed all previous instructions that S depends on. For instance, to execute $X = a$, the processor must have executed the last instruction that assigns into the variable a.

Out-of-order execution can explain the weak behavior of SB, but it can also explain the weak behavior of the "load buffering" program below (which is not allowed by x86-TSO).

$$\begin{array}{l} S_1 : \ a = X; \quad // \, 1 \\ S_2 : \ Y = 1; \end{array} \Bigg\| \begin{array}{l} S_3 : \ b = Y; \quad // \, 1 \\ S_4 : \ \mathsf{if}(b) \\ \qquad X = 1; \end{array} \tag{LB}$$

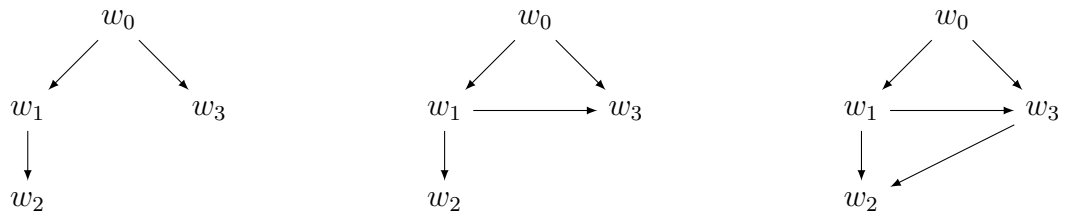
Instruction S_2 can be issued before S_1 because it does not depend on it. In contrast, S_4 cannot be issued before S_3 because its execution depends upon S_3 reading 1. The annotated outcome $a = b = 1$ of LB can be generated by executing the instructions in the following order: S_2, S_3, S_4, S_1.

Non-Multi-Copy Atomicity (non-MCA) Another subtle feature PowerPC follows is non-multicopy atomicity; a write access is observed by different threads at different times. For example, in the IRIW program the write $X = 1$ is observed by the second thread before the third thread. Similarly, the write $Y = 1$ is observed by the third thread before the second thread. As a result, the outcome $a = c = 1 \wedge b = d = 0$ is allowed by PowerPC unlike x86-TSO.

Coherence PowerPC ensures coherence; that is, for each location the memory accesses follow a total order consistent with the program order. To reason about coherence, Sarkar et al. [64] proposed the *coherence-by-fiat* model. In this model, the memory hierarchy is abstracted in *memory-subsystem*, where writes on each location are in partial order. The total order of the writes are decided based on the reads which read from these writes. For example, consider the following program.

$$\begin{array}{l} a = X; \quad // \, 1 \\ b = X; \quad // \, 3 \\ c = X; \quad // \, 2 \end{array} \Bigg\| \begin{array}{l} X = 1; \\ X = 2; \end{array} \Bigg\| \ X = 3;$$

Consider that the program has an execution where $a = 1 \wedge b = 3 \wedge c = 2$. In this execution we denote the writes $X = 0$, $X = 1$, $X = 2$, $X = 3$ by w_0, w_1, w_2, w_3 respectively. (a) is the initial partial order among the writes. At this point X reads value 1 on a. Next, X reads value 3 on b which appends an order w_1 to w_3 as shown in (b). Finally in (c), the read of X on c introduces the order w_3 to w_2. Along with the orders among the writes, the reads are also ordered and as a result the memory accesses on X a total order in this execution.



Fences and Cumulativity Apart from memory accesses, PowerPC has fences (a.k.a. barriers) to order the memory accesses. PowerPC has three types of fences: sync, lwsync, and isync. sync fences work as full barriers; memory accesses cannot move before or after a sync fence. lwsync is a lightweight fence; a lwsync orders a pair of independent memory accesses except store-load pairs. isync imposes the least ordering restriction; an isync following a conditional branch orders a pair of independent pair of load operations.

In addition to ordering independent memory access pairs in a thread, fences also have cumulative effect on ordering on memory accesses across threads. The effect of cumulativity is categorized as *A-cumulativity* and *B-cumulativity*. We explain these two concepts with the following programs taken from Alglave et al. [8].

$$X = 1; \;\Big\|\; \begin{array}{l} a = X; \quad \text{// 1} \\ \text{lwsync}; \\ Y = 1; \end{array} \qquad \text{(A-C)} \qquad\qquad \begin{array}{l} X = 1; \\ \text{lwsync}; \\ Y = 1; \end{array} \;\Big\|\; \begin{array}{l} a = Y; \quad \text{// 1} \\ Z = a; \end{array} \;\Big\|\; \begin{array}{l} b = Z; \quad \text{// 1} \\ A = b; \end{array} \qquad \text{(B-C)}$$

A-cumulativity orders the accesses across fence operations. For example, in the A-C program, if $a = X$ in the second thread reads from $X = 1$ in the first thread, it establishes an A-cumulative order from $X = 1$ to $Y = 1$ in that execution. B-cumulativity orders accesses before a read operation to the accesses after a read operation. Consider the B-C program. If $a = Y$ in the second thread reads from $Y = 1$ of the first thread and $b = Z$ reads in the third thread reads from $Z = a$ of the second thread then there is a B-cumulative order from $X = 1$ to $a = Y$, $Z = a$, $b = Z$, $A = b$ accesses in that particular execution.

Propagation Propagation decides the order in which writes accesses are propagated to the other threads. The propagation order does not contradict the coherence order and in addition orders writes on different locations. The propagation order is derived from fences, and other orders such as modification order (mo), that is, total order on same-location write accesses. For example, consider the following program.

$$\begin{array}{l} X = 2; \\ \text{lwsync} \\ Y = 1; \\ a = Y; \quad \text{// 2} \end{array} \;\Big\|\; \begin{array}{l} Y = 2; \\ \text{lwsync} \\ X = 1; \\ b = X; \quad \text{// 2} \end{array}$$

Consider the execution where $a = b = 2$ at the end of the program. In that execution $a = Y$ reads from $Y = 2$ which implies modification order from $Y = 1$ to $Y = 2$. In that case there is a propagation order from $X = 2$ to $X = 1$ and hence $b = X$ cannot read from $X = 2$ as in that case propagation order contradicts modification order. Hence the execution in forbidden in Power.

Consistency The validity of a PowerPC execution is checked against the PowerPC consistency conditions. The conditions are as follows.

- (SC-PER-LOC) There is a total order on the memory accesses on a particular location. Essentially this condition ensures that the execution is coherent.

- (NO-THIN-AIR) It requires the *happens-before* relation to be acyclic where happens-before is derived from preserved-program-order, read-from, and fences. This restriction forbids undesirable 'out-of-thin-air' behavior such as $a = b = 1$ in CYC program in Figure 2.5.

- (OBSERVATION) This condition constraints the set of writes from which a read operation can read. The condition requires that the combination of *from-read*, *propagation*, and *happens-before* relations to be acyclic.

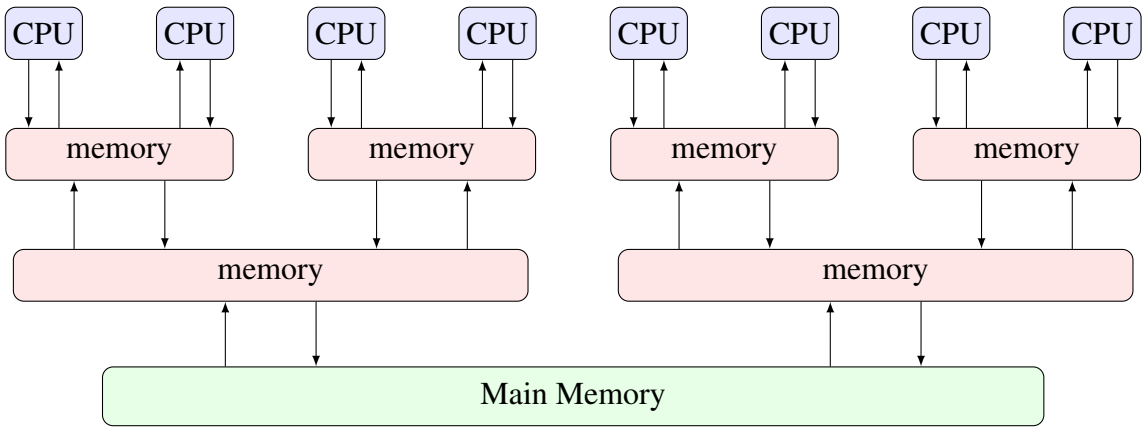
- (PROPAGATION) In a valid execution the propagation order should not contradict the modification order.

- (ATOMICITY) PowerPC implements atomic operation by *load-linked* (LL) and *store-conditional* (SC) instruction pair. This condition ensures that atomicity is preserved in a valid PowerPC execution.

The concurrency semantics of PowerPC is formalized by a number of papers. Batty et al. [12] and Sarkar et al. [64] proposed an operational model which is closer to the actual architecture. Later, Alglave et al. [8] proposed another operational operational model which abstracts away certain architectural details. Alglave et al. [8] also proposed an axiomatic model of PowerPC and demonstrate the equivalence between the operational and the axiomatic models. In our work we follow the Power axiomatic model described by Alglave et al. [8] and a stronger variant of that model, the *SPower* model, proposed by Lahav and Vafeiadis [35].

2.1.3 ARM

ARM architecture also demonstrates relaxed memory concurrency. The concurrency semantics results from the hierarchical organization of memory (captured by the *flowing* model) as follows.



We discuss about the concurrency semantics of ARMv7 and ARMv8; two versions of ARM architecture.

ARMv7 The concurrency semantics of ARMv7 has many similarities to PowerPC architectures with subtle differences. ARMv7 also has load, store, LL/SC, and fence instructions similar to the PowerPC. However, ARMv7 has no equivalence of the lwsync in PowerPC. Considering the formal semantics, ARMv7 has same set of consistency constraints as PowerPC, however, with subtle difference [8]. In PowerPC a write-after-read on the same location constitute in preserved program order unlike ARMv7. As a result, in the following program $a = 1$ is a possible outcome in ARMv7 unlike PowerPC.

$$\begin{array}{l} a = X; \quad // 1 \\ X = 1; \end{array} \Bigg\| \; Y = X; \Bigg\| \; X = Y; \qquad \text{(ARM-Weak)}$$

ARMv8 ARMv8 is a more recent architecture specification [9] compare to ARMv7. Flur et al. [27] propose the 'flowing' and 'POP' operational models for ARMv8 concurrency. The flowing model is based on hierarchical storage subsystem where written values flow from the processors to the memory hierarchy and during the read a processor reads the value from the closest memory. The flowing model considers various architectural components and closer to the actual architecture. The other model, partial-order propagation (or POP), abstracts the storage subsystem and in POP model all hardware threads are symmetrical.

More recently [9] is revised where ARMv8 concurrency semantics is simplified. The simplified model removes non-multicopy atomicity (non-MCA) along with other subtle changes. For instance, consider the following example:

$$X[0] = 1; \Bigg\| \begin{array}{l} a = X[0]; \quad // 1 \\ b = Y[a * 0]; \quad // 0 \end{array} \Bigg\| \begin{array}{l} c = Y[0]; \quad // 1 \\ d = X[c * 0]; \quad // 0 \end{array} \Bigg\| \; Y[0] = 1; \qquad \text{(IRIW+addr)}$$

The behavior in question is $a = 1, b = 0, c = 1, d = 0$. ARMv7 allows the program behavior due to non-multicopy atomicity whereas ARMv8 disallows this behavior.

Pulte et al. [61] have studied and formalized the ARMv8 concurrency which propose both operational and axiomatic semantics for ARMv8. The operational semantics is termed as *Flat model* which is a simplified version of the flowing model [27]. Pulte et al. [61] also define an axiomatic semantics for the ARMv8 revised semantics which matches the operational flat model with certain exceptions.

2.2 Dependencies

So far, we have discussed a number of hardware memory models. It is evident that the program behaviors in these architectures result from out-of-order executions and cannot be explained by interleaving executions. However, these out-of-order executions preserve certain *dependencies* among the memory accesses. Thus dependencies among the memory accesses play an important role in these relaxed memory behaviors.

We discuss the possible dependencies observed in the programming context. Dependencies can be categorized as data, address, output, anti and control dependencies as shown below.

$a = X;$	$a = X;$	$X = 1;$	$a = X;$	$a = X;$
$Y = a;$	$b = Y[a];$	$X = 2;$	$X = 1;$	$\mathsf{if}(a)\ Y = 1;$
(a) data-dep.	(b) address-dep.	(c) output-dep.	(d) anti-dep.	(e) control-dep.

When the value read by one instruction is used in another instruction, we have a *data dependency*. For example, in the program (a), the definition of Y is dependent on X as the value of X flows to Y through a. We cannot reorder these two instructions, because otherwise Y would get an older value of a.

A special case of data dependence are *address dependencies*, where the read value is used to determine the address of a memory location of a following instruction. For example, in program (b), the value returned by the the load of X determines the offset in the Y array.

If two instructions write to the same location, as in program (c), then these instructions can similarly not be reordered; they are in *output dependence*.

An *anti-dependency* occurs when a location is read before it is written to. Consider the program (c) where X is first read and then written to. One cannot simply reorder the two instructions, because then the read will return the wrong value for X. We note, however, that ARM partially relaxes anti-dependencies. In program (d), for example, it may propagate the later write to X before resolving the read of X; it just ensures that the read of X will eventually be resolved to return the value of some earlier write to X.

A *control dependency* between two instructions A and B occurs when the execution of B depends on the evaluation of a conditional expression dependent on A. For example, in program (e), instruction $Y = 1$ is control dependent on the load of X.

2.3 Explaining Behaviors in Relaxed Memory Models

Now we move on to the different styles of defining concurrency models. We have three main approaches:

- the *transformational* approach, which explains certain relaxed memory behaviors by program transformation;
- the *operational* approach, which explains the possible executions with some abstract machine;
- and the *axiomatic* or *declarative* approach, which places a number of constraints (axioms) on the allowed program executions.

2.3.1 Transformational Approach

We can explain a number of behaviors of relaxed memory programs considering the transformations performed at the compiler or hardware levels. As discussed earlier, many relaxed memory behaviors take place as a result of an out-of-order execution. An out-of-order execution can be interpreted as the reordering transformation of independent accesses. Thus, one

approach is to explain the relaxed behaviors is by transformations [35]. Given a program, first, we apply a set of allowed transformations and next, we explore the possible interleaving executions to check if a certain outcome is possible. For instance, in the LB program $a = b = 1$ is a valid outcome which can be explained by reordering followed by interleaving as follows.

$$\begin{array}{l} a = X; \\ Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ \mathsf{if}(b) \\ \quad X = 1; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} S_1 : \ Y = 1; \\ S_2 : \ a = X; \end{array} \Bigg\| \begin{array}{l} S_3 : \ b = Y; \\ S_4 : \mathsf{if}(b) \\ \quad S_5 : \mathsf{X} = 1; \end{array}$$

Then, consider an interleaving S_1, S_3, S_4, S_5, S_2 where load of Y in the second thread reads from the store of S_1 and load of X reads from the store of S_5 which results in $a = b = 1$.

Similarly, we can explain the $a = b = 0$ outcome in the SB program where if the reads happen before the writes in an interleaving then $a = b = 0$ is possible.

Lahav and Vafeiadis [35] have shown that the x86-TSO model (Total Store Order model for x86 architecture) can be explained by a set of reordering and elimination transformations followed by interleaving execution. They have also proposed a stronger model SPower [35] for PowerPC architecture which along with a set of reordering and elimination transformations captures PowerPC behaviors proposed by Alglave et al. [8].

These results facilitate the compilation proofs from the programming language consistency models to the architecture models. Given a higher level memory model we prove the correctness of various transformations and then we prove the compilation correctness for a stronger target model.

Limitations of Transformational Approach Though useful in many cases, transformational approach suffer from certain limitations. The approach depends heavily on allowed transformations and dependence analysis. For example, the $a = 1$ outcome in the ARM-Weak program cannot be explained by any valid reordering followed by interleaving execution. It is because the statements in the first thread establish an anti-dependence. However, if we consider thread sequentialization ($\mathbb{C}_1 \parallel \mathbb{C}_2 \rightsquigarrow \mathbb{C}_1; \mathbb{C}_2$) transformation, then we can a sequence of transformation which results in an execution where $a = 1$ is possible.

$$\begin{array}{l} a = X; \\ X = 1; \end{array} \Bigg\| \ Y = X; \Bigg\| \ X = Y; \quad \underset{\rightsquigarrow}{(1)} \quad \begin{array}{l} X = Y; \\ a = X; \\ X = 1; \end{array} \Bigg\| \ Y = X; \quad \underset{\rightsquigarrow}{(2)}$$

$$\begin{array}{l} X = Y; \\ a = Y; \\ X = 1; \end{array} \Bigg\| \ Y = X; \quad \underset{\rightsquigarrow}{(3)} \quad \begin{array}{l} X = Y; \\ X = 1; \\ a = Y; \end{array} \Bigg\| \ Y = X; \quad \underset{\rightsquigarrow}{(4)} \quad \begin{array}{l} S_1 : X = 1; \\ S_2 : a = Y; \end{array} \Bigg\| \ S_3 : Y = X;$$

In the first transformation the third thread is sequentialized with the first thread. Next, read-after-write (RAW) elimination replaces $a = X$ by $a = Y$ and as a result the anti-dependence turns out to be a *false*. In the next step we reorder $X = 1$ and $a = Y$ and then overwritten write elimination along with dead code elimination eliminate $X = Y$. In the resulting program, we can have an interleaving execution S_1, S_3, S_2 where Y and X reads from concurrent writes of X and Y respectively which results in $a = 1$ outcome.

Considering the shown limitation, the correct approach would be to define a formalization which provide a mathematical framework to reason about relaxed memory programs without any assumption on the set of allowed transformations. The correctness of desired program transformations has to be evaluated separately on the defined formalization.

2.3.2 Operational Approach

The operational approach defines an abstract machine which executes program statements step by step following the semantic rules. In such an abstract machine an execution consists of a set of states. A state is the combination of the memory locations along with respective values and a subset of location, value pairs comprise the state behavior. In addition, the abstract machine also has state transition for one or more execution steps. Finally the behavior of the final state of an execution denotes the outcome of that execution.

The operational approach is used to formalize concurrency semantics in both architecture as well as programming languages. For example, Owens et al. [54] define an operational model for x86, while Batty et al. [12] and Sarkar et al. [64] develop operational model for PowerPC. In case of ARM architecture, there are several operational models: Flowing, POP [27], and Flat [61].

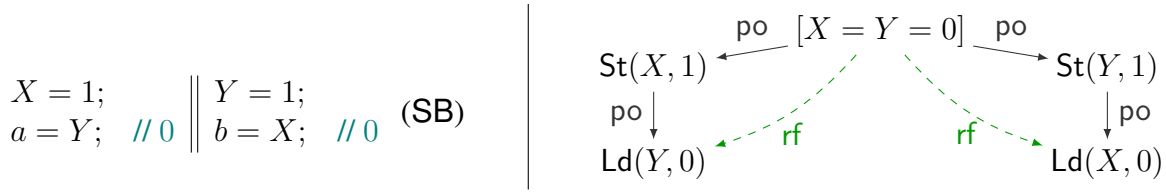
In the programming languages Nienhuis et al. [51], Kang et al. [33] proposed operational semantics for C11. We discuss the promising semantics proposed by Kang et al. [33] in more detail in Chapter 4.

While operational semantics techniques facilitate incremental constructions which is simpler and intuitive, there are certain issues in defining relaxed memory concurrency in operational models. First, programmers have to be aware about the underlying details of the memory organizations in the architectures. For example, Owens et al. [54] introduces write buffer in defining x86 semantics, Batty et al. [12], Sarkar et al. [64] introduce memory subsystem to formalize PowerPC concurrency. Moreover, out-of-order execution pose another challenge to operational models as syntactic order cannot guide the construction of an execution. For example, operational semantics cannot justify $a = b = 1$ outcome of the LB program. To avoid this limitation, Nienhuis et al. [51] proposed a different ordering of construction steps for the C11 model. On the other hand, promising semantics [33] introduces two constructs; timestamps and *promise* to justify C11 behaviors, such as $a = b = 1$ outcome of the LB program.

Even with these introduced constructs, however, operational models do not address all the concerns of relaxed memory concurrency. Rather the reasoning of these introduced constructs turn out to be significantly difficult. For instance, the C11 to PowerPC compilation proof proposed by Batty et al. [12] is later found flawed [8]. Nienhuis et al. [51] does not capture all the relaxed behaviors of C11 and proposed certain strengthening of the model. Apart from justifying certain behaviors, they do not provide DRF guarantee or prove any result on compilation correctness to the architectures. Kang et al. [33] provides a number of significant results but unfortunately do not address all the concerns and is extremely inflexible. We discuss more details of promising semantics [33] in Chapter 9.

2.3.3 Axiomatic/Declarative Approach

Axiomatic semantics applies a set of constraints to check whether a given execution is correct instead of constructing the execution. The constraints are applied on a set of program events and certain relations between these events. In case of relaxed memory concurrency the events represent the shared memory accesses in the execution there are a number of relations among these events. The events and relations in an execution is represented as a graph where the nodes represent events and the edges represent the relations. For example, the execution of the SB program where $a = b = 0$ can be represented as follows.



In this graph $[X = Y = 0]$ denotes that X and Y are zero-initialized. The $\mathsf{St}(X, 1)$, $\mathsf{St}(Y, 1)$ events denote the store accesses where X and Y are assigned values 1 respectively. The $\mathsf{Ld}(X, 1)$ event read event denote that read of X returns value 1 and $\mathsf{Ld}(Y, 1)$ event denote that read of Y returns value 1 respectively. Program-order (po) is a relation between a pair of events which captures the order in which the shared memory accesses have taken place. One obvious constraint in any concurrency model is that the program-order is irreflexive for a valid execution. Another basic relation is read-from (rf) which connects a write and a read event where the read reads the value written by the write event.

Now a memory model specifies a set of constraints which we apply on this graph to check the validity of the execution. For example, we can introduce a new order SC among events and enforce sequential consistency by a constraint:

> SC *is a strict total order on the events and does not contradict the combination of* po *and* rf *relations, that is,* $(\mathsf{po} \cup \mathsf{rf})^+$.

The execution graph of SB program shown above does not have such an SC order and hence this execution is invalid under sequential consistency.

As we have seen, axiomatic semantics abstracts away many details such as execution state, machine architecture and so on and provide a mathematical framework for memory model

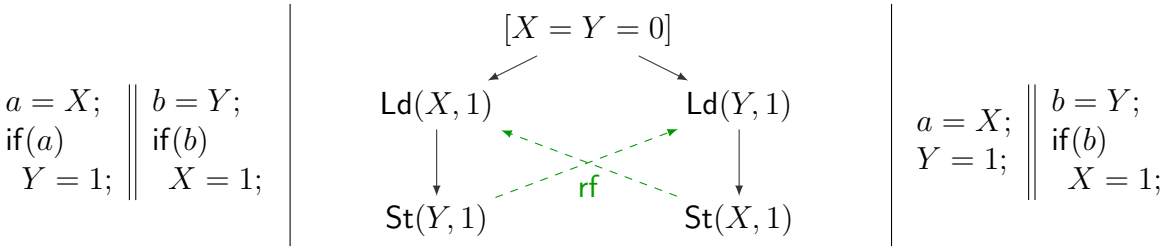


Figure 2.2: Outcome $a = b = 1$ in CYC and LB programs.

specification. Hence axiomatic approach has been used to specify a number of relaxed memory models, for both architectures such as Power [8] as well as programming language such as C11 [11, 74, 37].

Traditionally axiomatic models specify a set of constraints to ensure the validity of a single execution. However, the constraints cannot always differentiate between two underlying programs [13]. For example, the execution in Figure 2.2 which allows $a = b = 1$ in LB program also allows $a = b = 1$ in CYC program. Such an execution results in 'out-of-thin-air' behavior in CYC program; a behavior which cannot happen in any actual execution of CYC program.

2.4 Consistency Models in Programming Languages

Modern programming languages have adopted weak consistency models which provide platform independent abstractions. The relaxed memory models in programming languages differ from hardware models in certain ways.

2.4.1 True and False Dependence

In §2.2 we studied a number of dependencies which can be identified by syntactic analysis. However, syntactic analysis cannot always identify dependencies and hence we categorize dependencies as *true* or *false* dependencies.

A *true dependency* implies that indeed there is a dependency between a pair of instructions. In case of *false* dependence the dependency can be removed once we perform certain transformations. For instance consider the following program.

$$\begin{array}{l} a = X; \\ Y = a * 0; \quad // \; always\; 0 \end{array}$$

In this program though the definition of Y syntactically depends on X, we know that value of Y is always zero irrespective of the value of X. Hence the definition of Y has false dependence on the use of X. In this case, if a transformation removes the false dependence then the instructions can be reordered or out-of-order execution may take place.

Thus the interplay between dependencies and transformations plays a major role in deciding possible program behaviors.

The programming language memory models handle them in different manner from hardware models. Programming language consistency models intend to get rid of false dependencies to achieve further optimizations. Consider another example involving control dependency.

$$\begin{array}{l|l} a = X; \quad // \; 1 & \\ \mathsf{if}(a) & \\ \quad Y = 1; & b = Y; \quad // \; 1 \\ \mathsf{else} & X = b; \\ \quad Y = 1; & \end{array} \qquad \text{(LBfd)}$$

In the first thread of the LBfd program, the stores to Y are control-dependent on the load of X. However, it is a false dependence as in both paths of the conditional same value is stored on Y. Such false dependencies can be optimized away as follows and in sequence $a = b = 1$ is a legitimate outcome of the program in an interleaving.

$$\begin{array}{l} a = X; \\ \mathsf{if}(a) \\ \quad Y = 1; \\ \mathsf{else} \\ \quad Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ X = b; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a = X; \\ Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ X = b; \end{array}$$

On the other hand, often the hardware memory models do not differentiate between such false and true dependencies and hence disallow $a = b = 1$ outcome in this program.

2.4.2 Data Race

Another important concept in programming language concurrency models is data race. In an execution of a program if multiple threads access a shared location concurrently and at least one of the access is a write then the execution has data race. A data race is problematic as two racy write operations may spoil the value of one location or a racy read may return an arbitrary value. As a result, often the racy executions have undesired outcome and thus data races are often a major source of concurrency bugs in the program. Hence data race freedom is considered an important property in relaxed memory models, as there are certain guarantees for race free programs ensured by data race freedom theorems.

Now we discuss various relaxed memory programming language models. We start our discussion with Data-Race-Free-0 (DRF0) model which is the basis for more recent C11 and Java relaxed memory models.

2.4.3 Data-Race-Free-0 (DRF0)

The Data-Race-Free-0 or DRF0 model is proposed by Adve and Hill [7]. In DRF0 model the variables and their respective accesses are either of synchronization types or of data types. To enable the synchronization on hardware, the underlying hardware must have certain instructions to recognize and enable synchronization operations. In addition, a synchronization operation must access a single location. When a synchronization variable reads-from a synchronization write then it establishes synchronization (sw). The synchronization along with program order (po) establishes a *happen-before* (hb) order between the events, that is, $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+$. The events which are not in happens-before relation are concurrent events and concurrent events on the same location are called *conflicting* accesses. If one of the conflicting accesses is a write then it is defined as a data race. As the synchronization variables facilitate communication across threads, data races on synchronization locations are acceptable. However, data races on data variable are undesirable in an execution and hence the behavior of a racy program is undefined. Thus DRF0 model defines the semantics of programs

with enough synchronization, that is, the programs which do not have any racy executions. The DRF0 model also provides DRF-SC (sequential consistent) guarantee; a race-free DRF0 execution demonstrate no weaker behavior than sequential consistency. Considering correct program transformations, it is safe to eliminate redundant data accesses or reorder independent data accesses on different locations. The data accesses can safely move before synchronization writes or after synchronization reads. However, the model does not allow elimination of (redundant) synchronization accesses or reordering of synchronization access pairs.

In more recent times the DRF0 model has been extended to define the Java and C11 concurrency semantics. Similar to the synchronization accesses, Java introduces *volatile* accesses and C11 introduces *atomic* accesses. In addition, similar to the data accesses the Java and C11 models have plain or *non-atomic* accesses.

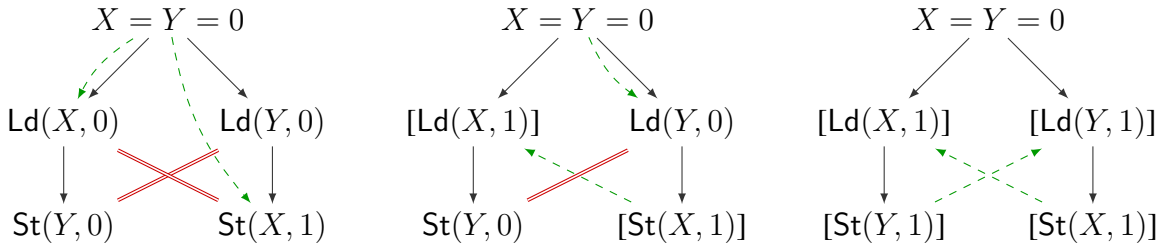
2.4.4 Java

Java is the first language to introduce relaxed memory concurrency semantics in the programming language by extending the DRF0 model. The model has gone through a number of revisions. Gosling et al. [28] proposed the first specification of Java concurrency and Pugh [60] identified a number of flaws in this model. Manson et al. [46] proposed a new model which ensures DRF guarantee, avoid OOTA behaviors, and allows reordering transformations. However, Cenciarelli et al. [20] identified a flaw in this model [46] which was later fixed by Aspinall and Sevcík [10]. The latest specification is available in [2].

Primitives The concurrency primitives in Java consist of *lock/unlock* operations along with 'normal' and 'volatile' accesses. The 'lock/unlock' and 'volatile' operations are synchronization actions performed during an execution.

Executions In the Java semantics [2], an execution includes events along with *program-order* (po) and *synchronization order* (so) relations among the events. Program order denotes the syntactic order among the events following the intra-thread semantics. Synchronization primitives establish *synchronization order* (so) when (1) an 'unlock' event synchronizes with all subsequent 'lock' operations, or (2) a volatile read reads-from a volatile stores. Similar to the DRF0 model *program-order* (po) and *synchronization order* (so) establishes a *happens-before* (hb) relation, that is, $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{so})^+$. Moreover, if two events access same location where at least one of the accesses is a write then these events are in conflict. If two conflicting events are not in happens-before relation then it results in data race. In addition, an execution also has *write-seen* function W such that given a read event r, $W(r)$ returns the write from which r reads its value. An execution is happens-before consistent when no read r in the execution reads from a write $W(r)$ which happens after the read. Moreover, there exists no intervening write w' such that $\mathsf{hb}(W(r), w')$ and $\mathsf{hb}(w', r)$.

Race-Free Programs If all sequentially consistent (SC) executions of a Java program are race-free, we say that the program is well synchronized. In this case, the program has no other



(a) $a = b = 0$ (b) $a = 1 \wedge b = 0$ (c) $a = b = 1$

Figure 2.3: Execution of the example by Sevcík [65].

weak executions under the Java memory model. In other words, Java concurrency satisfies the DRF-SC guarantee: well-synchronized programs exhibit only SC behavior.

Racy Programs When a program lacks enough synchronization, then the program may exhibit data races in certain executions. Java concurrency defines semantics for racy programs. The semantics uses the concept of *commit* and *re-execute* to justify a racy execution. We explain the notion with an example taken from Sevcík [65].

$$\begin{array}{ll} 1 : a = X; & \text{// } 1 \\ 2 : Y = a; & \end{array} \Bigg\| \begin{array}{ll} 3 : b = Y; & \text{// } 1 \\ 4 : X = 1; & \end{array}$$

A valid execution in this program is $a = b = 1$; it is possible as the instructions in the second thread can be reordered followed by an interleaving $4, 1, 2, 3$.

Java justifies this outcome as shown in Figure 2.3. First, it generates execution 2.3a, where the loads read from the initialization writes. This execution has data races both on X and on Y. In execution 2.3b, we restart the execution and choose a race—the race on X in this case. The execution commits the race on X where $\mathsf{Ld}(X, 1)$ reads from $\mathsf{St}(X, 1)$. The other read $\mathsf{Ld}(Y, 0)$ reads from a happens-before write. As a result, the execution results in $a = 1 \wedge b = 0$. Next, we restart the execution in 2.3c and by resolving the race on Y so that $\mathsf{Ld}(Y, 1)$ reads from $\mathsf{St}(X, 1)$.

Coherence Java does not enforce coherence (i.e., the SC-per-location property). There are programs with only one shared location, for which Java exhibits non-SC behavior. Consider, for example, the following program from [45]:

$$\begin{array}{ll} a = X; & \text{// } 2 \\ X = 1; & \end{array} \Bigg\| \begin{array}{ll} b = X; & \text{// } 1 \\ X = 2; & \end{array} \qquad \text{(RW2Loc)}$$

The non-SC outcome $a = 2 \wedge b = 1$ is allowed by the Java memory model.

2.4.5 C11

C/C++ extended the DRF0 model to define its relaxed memory concurrency semantics. The semantics is discussed in Boehm and Adve [16] and later specified in the 2011 ISO C/C++ standards [30, 29] (termed as C11).

Along with, C11 specifies a set of memory consistency constraints. These constructs provide a higher-level platform-independent abstraction over the concurrency semantics of existing multi-core hardware implementations. The programmers use these constructs to write programs with required consistency guarantees and these C11 constructs are mapped to the instructions in the target multicore hardware in optimal fashion. The C11 model, similar to the architectures, also intend to allow various shared memory transformations (e.g. reordering, elimination etc) based on the access types and memory orders.

Primitives The memory access operations in C11 are *load* (Ld), *store* (St), *update* (U) such as compare-and-swap, fetch-and-add and so on. In addition to the memory accesses, C11 has *fence* ($\mathcal{F}$) as a shared memory operation as it affects the concurrency semantics. The load and store operations can be either atomic or non-atomic, and the update and fence operations are atomic operations. The shared memory operations (load, store, update, fence) are annotated with a *memory order*, o. For loads, this can be non-atomic (NA), relaxed (RLX), acquire (ACQ), or sequentially consistent (SC). For stores, it can be non-atomic, relaxed, release (REL), or sequentially consistent. For update, it can be relaxed, acquire, release, acquire-release (ACQ-REL) or sequentially consistent. Finally, a fence can be of release, acquire, acquire-release or sequentially consistent In increasing strength, these orders are: $\text{NA} \sqsubset \text{RLX} \sqsubset \{\text{ACQ}, \text{REL}\} \sqsubset \text{ACQ-REL} \sqsubset \text{SC}$. Thus the concurrency primitives in C11 are as follows.

$$\mathsf{Ld}_{(\text{NA}|\text{RLX}|\text{ACQ}|\text{ACQ-REL}|\text{SC})} \mid \mathsf{St}_{(\text{NA}|\text{RLX}|\text{REL}|\text{ACQ-REL}|\text{SC})} \mid \mathsf{U}_{(\text{RLX}|\text{ACQ}|\text{REL}|\text{ACQ-REL}|\text{SC})} \mid \mathcal{F}_{(\text{ACQ}|\text{REL}|\text{ACQ-REL}|\text{SC})}$$

The load or update accesses are *read* accesses, and the store or update accesses are *write* accesses. Moreover, the accesses with release or stronger memory orders are termed as 'release accesses' and similarly the operations with acquire or stronger memory accesses are termed as 'acquire accesses'.

Semantics C11 introduces a set of relations among the shared memory accesses. Program-order (po) captures the syntactic order among the intra-thread shared memory access. Reads-from (rf) relation relates a pair of write and read accesses. When a read access reads from a write access then it establishes a reads-from (rf) relation between the pair of write and read accesses. Modification order (mo) establishes order on the per-location write operations. Finally, the SC accesses are ordered by sequential consistency (SC) order.

Based on these primitives relations C11 defines a number of derived relations such as synchronization order (sw), happens-before (hb) and so on. The synchronization-with relation is establish between a pair of release acquire accesses. For instance, when an acquire read reads-from a release write then it establishes synchronization order (sw). The combination of synchronization order and program order results in happens-before relation. These relations are used to define certain axiomatic constraints for C11 execution.

Consistency The axiomatic rules on the concurrency primitives define the consistency constraints which checks whether a C11 execution is valid. These constraints are of two categories.

(1) *Event orderings*. A number of constraints ensure that the accesses in an execution follow certain orders. Such orders constitute of one or combination of multiple relations.

(2) *Valid reads*. These constraints checks the validity of 'reads-from' relations, that is, given a pair of read and write accesses, whether the read can read from the write operation.

Data race Considering the ill-effect of data race, similar to Java memory model [2], in C/C++ semantics [30, 29] a program with racy execution has undefined behavior. However, according to C11 data race on atomic accesses are safe and has defined behavior. On the other hand, *non-atomic-race* implies concurrent write-write or write-read access pair on the same location where at least one of the access is non-atomic. An execution with non-atomic-race has no restriction on the outcome and in C11 a program with non-atomic-race has undefined semantics [16, 18]. Hence in programming practices, programmers must avoid writing programs with data race on non-atomics.

Program Transformations C11 concurrency primitives and the associated rules are general enough to allow both compiler optimization and code generations for the target architectures. The semantics intends to allow the reordering of independent concurrency primitives and elimination of redundant concurrency primitives. Moreover, the model supports efficient code generation [4] for a number of underlying architectures such as x86, PowerPC, ARM and so on.

Evolution of the C11 Concurrency Semantics

There are a number of attempts to formalize various aspects of C11 concurrency semantics [11, 73, 74, 14, 33, 37]. and the C11 model has gone through a number of revisions. The initial C11 consistency constraints are described in Batty et al. [11] and more concisely in Vafeiadis and Narayan [73]. However, this model suffers from certain limitations.

Non-Atomic Reads According to the C11 specification a non-atomic read reads only from a *happens-before* write. Vafeiadis et al. [74] demonstrate that this constraint turns out to be too restrictive for a number of desired compiler transformations such as roach motel reorderings, access strengthening and so on. Consider the example from Vafeiadis et al. [74]:

$$\begin{array}{|l|l|l|l|}
Z_{\text{REL}} = 1; & \textbf{if}\,(X_{\text{RLX}})\ \text{// } 0 & \textbf{if}\,(Y_{\text{RLX}})\ \text{// } 0 & a_{\text{NA}} = 1; \\
a_{\text{NA}} = 1; & \quad \textbf{if}\,(Z_{\text{ACQ}})\ \text{// } 0 & X_{\text{RLX}} = 1; & \rightsquigarrow\ Z_{\text{REL}} = 1; \\
 & \quad\quad \textbf{if}\,(a_{\text{NA}})\ \text{// } 0 & & \\
 & \quad\quad y_{\text{RLX}} = 1; & &
\end{array}
\quad\quad
\begin{array}{|l|l|}
\textbf{if}\,(X_{\text{RLX}})\ \text{// } 1 & \textbf{if}\,(Y_{\text{RLX}})\ \text{// } 1 \\
\quad \textbf{if}\,(Z_{\text{ACQ}})\ \text{// } 1 & X_{\text{RLX}} = 1; \\
\quad\quad \textbf{if}\,(a_{\text{NA}})\ \text{// } 1 & \\
\quad\quad y_{\text{RLX}} = 1; &
\end{array}$$

In the source program the only possible outcome is $Z = a = 1$ and $X = Y = 0$. After the roach motel reordering of the accesses pairs in the first thread a in the second thread reads from the store of a in the first thread. As a result, we can have the above mentioned execution in C11 where $X = Y = Z = a = 1$ is possible as an outcome. Thus, roach motel reordering is unsound in C11 when a non-atomic read is not allowed to read from a *concurrent* write.

Semantics of SC Accesses The C11 semantics supports SC memory order for the shared memory accesses and fences. There are SC relation among a pair of accesses having SC order and the SC relation enforces a total order on the SC accesses in a valid C11 execution. In a valid C11 execution with only SC accesses, the SC order denotes the interleaving order among the accesses. However, the interaction of SC accesses with non-SC accesses introduce subtle issues and has gone through a number of changes.

Strengthening constraint for SC *reads.* According to C11 semantics an SC read reads only from a SC write w which is immediately preceding in SC order to the same location or from a non-SC write which does not happen-before w.

However, Vafeiadis et al. [74] observe that this restriction is too strong and as result memory order strengthening is unsound considering the following example where the behavior in question is $r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$.

$$\begin{array}{l} X_{\text{RLX}} = 3; \\ Y_{\text{SC}} = 2; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X_{\text{SC}} = 3; \\ Y_{\text{SC}} = 2; \end{array} \quad \textbf{Context:} \left[\; - \; \middle\| \; \begin{array}{l} X_{\text{RLX}} = 1; \\ X_{\text{SC}} = 2; \\ Y_{\text{SC}} = 1; \end{array} \; \middle\| \; \begin{array}{l} Y_{\text{SC}} = 3; \\ r = X_{\text{SC}}; \end{array} \; \middle\| \; \begin{array}{l} s_1 = X_{\text{RLX}}; \\ s_2 = X_{\text{RLX}}; \\ s_3 = X_{\text{RLX}}; \\ t_1 = Y_{\text{RLX}}; \\ t_2 = Y_{\text{RLX}}; \\ t_3 = Y_{\text{RLX}}; \end{array} \right]$$

Following the SC constraint discussed above, in an execution of the source program whenever $s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$ holds then $X_{\text{SC}} = 2$ in the second thread is the immediate SC preceding write w.r.t read $r = X_{\text{SC}}$ in the third thread. However, in a consistent execution read $r = X_{\text{SC}}$ may read from $X_{\text{SC}} = 2$ or $X_{\text{SC}} = 3$ but not $X_{\text{RLX}} = 1$ as it always happens-before $X_{\text{SC}} = 2$. Hence this execution is not consistent.

Now, if we strengthen the $X_{\text{RLX}} = 3$ into $X_{\text{SC}} = 3$ in the first thread as shown above, then it establishes SC order from $Y_{\text{SC}} = 1$ to $X_{\text{SC}} = 3$ and hence $X_{\text{SC}} = 3$ is immediate SC order successor of read $r = X_{\text{SC}}$ on location X in the execution. Now reading 1 for $r = X_{\text{SC}}$ access is valid as $X_{\text{RLX}} = 1$ does not happens-before $X_{\text{SC}} = 3$. As a result, the execution is consistent and introduces the questionable behavior $r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$ in the target.

To avoid such problem Vafeiadis et al. [74] strengthens the constraint where it considers all SC preceding same-location writes instead of only 'immediate' preceding write on the same location. As a result, $r = X_{\text{SC}}$ reading from $X_{\text{RLX}} = 1$ is not valid anymore and the target program does not have the questionable behavior. Thus the modified SC read constraint preserves correctness of access strengthening transformation.

Removal of SC *order.* The SC order has been a primitive component in the original C11 model. However, [14] showed that SC order is not a primitive component; it is possible to derive SC ordering among the SC accesses using other relations. However, the new constraint proposed for SC accesses contains subtle flaws. Manerkar et al. [44] come up with the counter-examples to show that the proposed constraints by Batty et al. [14] are too strong to preserve compiler correctness from C11 to Power and ARMv7. Consider the following example where

initially $X = Y = 0$:

$$X_{\text{SC}} = 1; \left\| \begin{array}{l} a = X_{\text{ACQ}}; \quad // 1 \\ b = Y_{\text{SC}}; \quad // 0 \end{array} \right\| \left\| \begin{array}{l} c = Y_{\text{ACQ}}; \quad // 1 \\ d = X_{\text{SC}}; \quad // 0 \end{array} \right\| Y_{\text{SC}} = 1;$$

The outcome $a = c = 1 \wedge b = d = 0$ is restricted by C11 semantics proposed by Batty et al. [14], but when mapped to Power instructions then it is an allowed outcome. Moreover, the semantics does not ensure interleaving semantics even when there are SC fences between every shared memory access pairs.

Repairing SC *semantics*. Lahav et al. [37] addresses these concerns of SC semantics and proposed RC11 semantics. They prove the correctness of RC11 to Power and ARMv7 compilations. The proposed semantics also ensure that when we place SC fences between every pair of shared memory accesses, it restores interleaving behavior.

Out-of-Thin-Air Reads Consider the $a = b = 1$ outcome in CYC program. Hardware models take dependencies into account and restricts this outcome. However, hardware models also restrict $a = b = 1$ outcome in LBfd program due to *false* dependency as shown in §2.3.1. To perform optimizations C11 differentiates between *true* and *false* dependencies and hence the formal model cannot take dependency into account while defining a formal model. As a result, C11 allows $a = b = 1$ outcome in CYC program which is an out-of-thin-air behavior.

The original C11 model [11] is too relaxed to provide DRF-SC guarantee and eliminate OOTA executions [74]. For example, consider the CYC program from Figure 2.5.

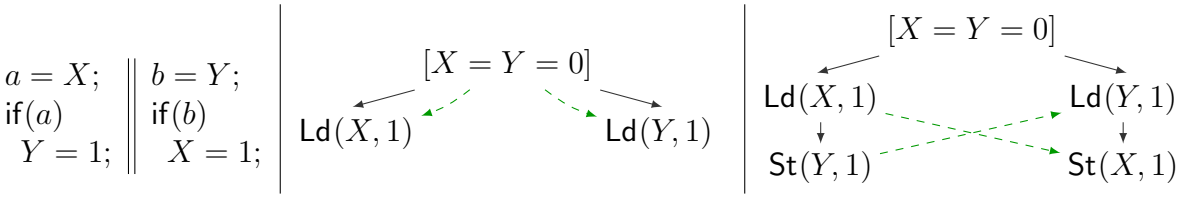


Figure 2.4: CYC program, only execution, and allowed OOTA execution in C11.

The only possible execution of this program is $a = b = 0$ and the program is non-racy. However, the C11 model allows $a = b = 1$ outcome in this program. This is an OOTA behavior as no actual execution demonstrate this outcome. Moreover, this outcome demonstrates an weaker behavior even when the program is non-racy under interleaving/SC execution. Thus, due to 'relaxed' atomics, the C11 model [11] does not provide DRF-SC guarantee. According to C11 specification [30, 29] this behavior is undesirable but allowed. Vafeiadis et al. [74] evaluated two alternatives by restricting the happens-before (hb) and read-from (rf) cycles.

The first alternative is to restrict hb $\cup$ rf cycle which was earlier proposed by Vafeiadis and Narayan [73], Boehm and Demsky [17]. This constraint is known affect the compilation results; the load-store reordering is disallowed and the mapping of of relaxed reads in ARM/Power architectures requires to introduce either a lightweight fence or a bogus branch between every shared load, store pair [17].

The proposed second alternative in [74] is to restrict hb$\cup$rf cycle where source or destination of an rf edge is non-atomic. This solution provides many nice properties including the efficient

mapping to ARM/Power where a shared load, store pair does not require any in-between fence or bogus branch. However, this solution disallows the reorderings of non-atomic load and store accesses Vafeiadis et al. [74].

These analyses demonstrate the limitations of per-execution based C11 semantics in dealing OOTA behavior of CYC program without affecting optimal compilation.

To address this issue Kang et al. [33] proposed an operational semantics for a fragment of C11 primitives which eliminates the OOTA, provides DRF guarantees, and allows various transformations. Batty et al. [14] simplified the semantics of C11 sequential consistent accesses significantly. However, the model turned out to be too strict and hence later rectified by Lahav et al. [37]. While each of these approaches made significant progress to define a C11 relaxed memory semantics, none of these approaches has been successful to address all the concerns in a single model.

2.5 Consistency Models in Compilers

A compiler translates a program written in a particular programming language to an underlying architecture in a semantic preserving manner. Hence whiling compiling a concurrent program, a compiler has to be aware of the memory consistency models of both the programming language as well as the architecture. However, in state-of-the-art C11 compilers such as GCC, LLVM the compilation takes place in multiple steps. First, the front-end translates the C11 program to the compiler's intermediate representations (IR), then the compiler performs a number of optimizing transformations on the intermediate representation of the program, and finally generates the target code for the particular architecture. Thus the intermediate representations (IR) in compilers facilitate compiler optimizations and mappings to the architectures. To capture the semantics of high level languages features, compilers also introduce similar features in its IRs. In this spirit, compilers like LLVM also introduce relaxed memory concurrency primitives to support C11 compilation.

2.5.1 LLVM

The LLVM compiler [3] supports the compilation of C11 concurrent programs to the target architectures such as x86, PowerPC and so on. Moreover, LLVM applies a number of optimization passes on a program before it generates the target code. In order to support the C11 concurrency compilation and the optimizations, LLVM also introduced it's own concurrency semantics. The semantics is described informally in LLVM documentation [43]. LLVM concurrency semantics introduces similar set of primitives compared to that of C11 and applies similar set of rules. However, it differs from C11 in the treatment of non-atomic-races. Contrary to C11, according to LLVM semantics a program having non-atomic-race between write and read access pair have defined behavior and the racy read returns an 'undef' expression; a placeholder for an arbitrary constant value. We discuss the details of 'undef' and LLVM concurrency semantics in §5.3.2 along with the formal models and related results.

2.5.2 Transformation Correctness

A transformation is correct when the transformation preserves the semantics of the source program in the target program. The semantics or observable behavior of a program consists of the set of outcomes of the valid executions. In case of shared memory concurrent programs, an execution outcome is the final values on the shared memory locations. Note that, in shared memory concurrency the validity of an execution of a program is decided by the memory model; and as a result, the correctness of a transformation are affected by the memory models of the source and the target programs.

We consider two types of transformations in a compiler: (1) efficient mapping of shared memory primitives from one language to the other (2) transformations performed by various optimizations. In the mapping transformation the consistency model differs between the source and the target whereas during the optimization transformation the memory model remain same for the source and the target programs.

Mapping A mapping is optimal if there is no other mapping which results in more efficient target program preserving the transformation correctness. For instance, Beringer et al. [15] enlisted the desired mappings of all of the C11 shared memory primitives to various target architectures including x86 and PowerPC. Consider the C11 to the PowerPC mapping; the C11 non-atomic load, store operations are mapped to the PowerPC load and store instructions receptively. The C11 atomic accesses have memory orders - for instance, a store operation having *release* memory order is mapped to a store instruction proceeded by an lwsync fence. Instead of an lwsync, we can also use a sync fence, however it will not be an optimal mapping as lwsync is cheaper than sync in terms of performance yet preserves the transformation correctness. The mappings of Beringer et al. [15] are well accepted and often followed by the state-of-the-art compilers such as GCC and LLVM.

However, proving the correctness of the optimal mapping of the shared memory primitives pose non-trivial challenges. The correctness requires rigorous proofs considering the memory models of the source and the targets. For instance, Batty et al. [11] proved the mapping correctness from C11 shared memory primitives to x86 instructions. Lahav et al. [36], Lahav and Vafeiadis [35], Kang et al. [33], Lahav et al. [37] have proposed variations of the C11 and x86/PowerPC memory models and have proved the mapping correctness. They have shown that the proposed models correctly preserve the optimal mapping suggested by Beringer et al. [15].

Batty et al. [12], Sarkar et al. [64] proposed proofs of the mapping correctness of the C11 primitives to the PowerPC architecture. However, Manerkar et al. [44] found an error in the proof of Batty et al. [12] in handling the mix of SC and non-SC primitives. Recently, Lahav et al. [37] discussed various issues with the semantics of C11 sequentially consistent primitives and proposed fixes that achieve correct mapping of C11 to PowerPC.

Optimizations The common transformations performed on concurrent programs are reordering of independent memory accesses and elimination of redundant memory accesses. These transformations often take place as part of various optimizations in a compiler. For example, a compiler may perform reordering $X_{\text{REL}} = 1; Y_{\text{NA}} = 1; \rightsquigarrow Y_{\text{NA}} = 1; X_{\text{REL}} = 1;$ to

optimize a C11 program where X and Y are independent shared variables. A compiler may also eliminate redundant shared memory access, for example, $X_{\text{NA}} = 1; X_{\text{NA}} = 2; \rightsquigarrow X_{\text{NA}} = 2;$ where $X_{\text{NA}} = 1$ is a redundant access. Hence allowing these transformations is one of the major goals for a relaxed memory consistency model.

Burckhardt et al. [19] proved the correctness of a number of transformations for various hardware memory models. Sevcík [66] studied similar transformations on a relaxed model which consists of (SC) atomic and non-atomic accesses and lock/unlock having DRF guarantee and proved the correctness of these transformations. Similarly the set of valid transformation in the C11 model is studied by [48]. In [74], we show that the original C11 [11] does not allow various common transformations and hence proposed certain fixes for the C11. We proved correctness of the various transformations on C11 as well as the variance of C11. Lahav and Vafeiadis [35] as well as Kang et al. [33] study the optimization correctness for the various proposed C11 memory models. In [22] we have studied the correctness of subset of these transformations in a fragment of LLVM concurrency semantics.

2.6 Challenges

So far we observe that C11 specifies an axiomatic relaxed memory concurrency semantics which intends to enable desired compiler optimizations and efficient compilation to the x86, PowerPC, ARM architectures and we require to formalize the semantics to reason about program behaviors and compiler correctness. However, we also observe the shortcomings of existing formal techniques used for defining C11 formalization.

We already know that C11 allows $a = b = 1$ OOTA outcome in CYC program. To avoid the $a = b = 1$ in CYC, Lahav et al. [37] propose the RC11 model which strengthens the C11 model with the constraint that $(\mathsf{po} \cup \mathsf{rf})^+$ has to be acyclic in a valid execution. As a result, RC11 restricts the relaxed load-store pair reordering transformation in the compiler level and the model requires extra fences between a pair of load and store accesses to restrict the reordering at the hardware level. Both of these options affect performance for the C11 programs.

Thus C11 suffers from a long-standing tradeoff between efficient compilation and OOTA behavior. To address this tradeoff of a number of alternative approaches are proposed [33, 31, 55]. Kang et al. [33] addresses the OOTA issue by introducing auxiliary constructs like 'promise', 'fulfill' in the operational semantics for a fragment of C11 concurrency. On the other hand, Jeffrey and Riely [31], Pichon-Pharabod and Sewell [55] uses event structures to reason about multiple execution together to disallow the OOTA outcome in CYC program.

In this scenario it is a nontrivial to define a formal model for C11 concurrency which addresses all the concerns of (1) resolving OOTA behavior (2) providing DRF guarantee (3) allowing desired program transformations and efficient mappings to the architectures (4) handling all types of accesses of C11 concurrency. Going forward we define a formal model to address this long-standing challenge.

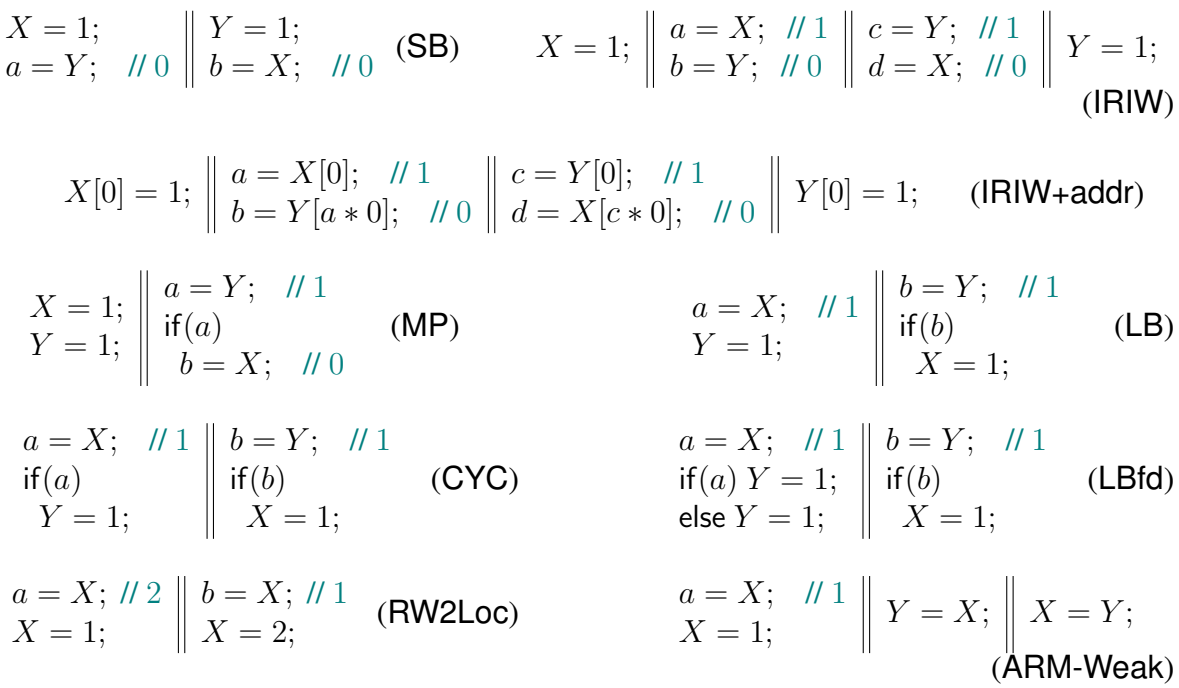


Figure 2.5: A collection of standard litmus tests. All locations are initialized to zero, and all accesses are plain (Java) or relaxed (C11).

2.7 Summary

In this chapter, we have discussed the prior work on weak memory models.

- The architectures (x86, PowerPC, ARMv7, ARMv8), programming languages (Java, C11), and compilers have their own memory consistency models. These models subtly differ from one another.
- Efficient transformation from the C11 to the target architectures is highly desirable. However, it is nontrivial to achieve correctness of these transformations considering the complexities of these memory models.
- We have seen the styles of formalizations of these memory models. We note that each of these approaches has its own advantages and disadvantages.
- Finally we point out to an important observation by Batty et al. [13] which demonstrates the limitation of 'per execution' based semantics models. We discuss how these semantic schemes suffer from a trade-off to choose between out-of-thin-air behavior and read-write transformation.

To demonstrate the subtle complexities of the memory models we have used some small programs along with particular behavior. These programs together with the respective behaviors are called litmus tests. In Figure 2.5, we consolidate the litmus tests that we have seen so

Table 2.1: Comparison of existing weak memory models on standard litmus tests.

	x86 [54]	Power [8]	ARMv7 [8]	ARMv8 [27]	ARMv8 [61]	Java [2]	C11 [11]	RC11 [37]
SB	✓	✓	✓	✓	✓	✓	✓	✓
MP	✗	✓	✓	✓	✓	✓	✓	✓
IRIW	✗	✓	✓	✓	✓	✓	✓	✓
IRIW+addr	✗	✓	✓	✗	✗	✓	✓	✓
ARM-Weak	✗	✗	✓	✓	✗	✓	✓	✓
RW2Loc	✗	✗	✗	✗	✗	✓	✗	✗
LB	✗	✓	✓	✓	✓	✓	✓	✗
CYC	✗	✗	✗	✗	✗	✗	✓	✗
LBfd	✗	✗	✗	✗	✗	✓	✓	✗

far, and in Table 2.1 we say whether the corresponding behaviors are allowed by the various concurrency models we have seen so far.

The table provides a quick visual comparison between the various models. For instance, x86 is the strictest among these models as it only allows the SB behavior and forbids all the other behaviors. All the other models, however, also allow the weak behavior of MP.

IRIW program behavior is allowed in all models except x86. Most models allow the weak behavior of IRIW+addr; only x86 and ARMv8-Flat are multicopy-atomic which forbid this behavior. ARMv7, ARMv8-Flowing/POP and the language-level models also allow the weak behavior of ARM-Weak, which is disallowed by x86, Power, and ARMv8-Flat.

The Java semantics [2] allows all the litmus tests which are allowed by the architectures. It also disallows the problematic behavior in CYC program. However, Java semantics has weak coherence guarantees and as a result allows RW2Loc program outcome which does not agree with any architecture.

The original C11 semantics allows all the litmus tests which are allowed by the architectures. Hence, C11 is an efficient architecture-independent memory semantics at the programming language level. Its main problem, however, is that C11 allows out-of-thin-air outcome in CYC program. The RC11 model [37] fixes this outcome, but due to its $\mathsf{po} \cup \mathsf{rf}$ acyclicity constraint, it also restricts the weak behaviors of LB and LBfd.

In the next chapter, we address the limitation of 'per execution' semantics by event structures that capture multiple executions. We use event structures to formalize an appropriate semantics for the C11 concurrency primitives.

3 The WEAKEST Memory Model

In §2.6 we have seen that the techniques which reason about individual executions of a program have a limitation: such a semantics either allows out-of-thin-air execution in a program or disallows read-write reorderings. To address this limitation we use *event structure* to formalize the program semantics.

In this chapter, we propose the WEAKEST or 'WEAK Event Structure' model based on event structures. Before discussing the formal details, we explain the basic ideas of our approach and how event structure can resolve the above mentioned trade-off between OOTA behavior and read-write reordering by distinguishing the executions of the CYC and LB programs.

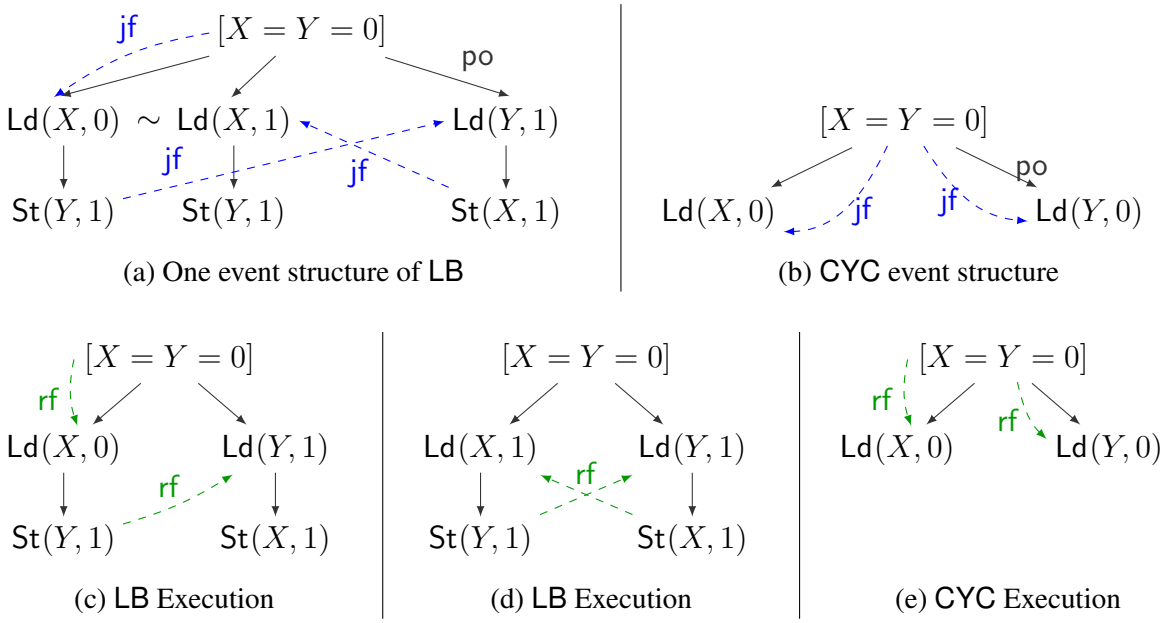
3.1 Justified Event Structures

An event structure is a representation of multiple executions of a program. It comprises a set of events representing the program's individual memory accesses and two relations over these events:

- The *program order* (po) represents the control flow order of the program; it relates two events a and b if they belong to the same thread and a is executed before b.
- The *conflict relation* (cf) relates events of the same thread that *cannot* belong to the same execution. A conflict, for example, occurs between two load events correspond to the same program instruction that return different values. By definition, cf is a symmetric relation. Moreover, when a and b conflict, and c is after b in program order, then a and c also conflict. That is, a conflict results in two different branches of the event structure, after which all events conflict with one another.

To these basic event structures, we add an extra relation, which we call the *justified-from* relation (jf). For each read event in the event structure, jf associates a (prior) write event that *justifies* the read returning its value. Namely, the write should have written the same value to the same location that the read is reading from.

Example Figure 3.1a demonstrates an event structure of LB program in graphical model where nodes denote the events, and edges denote relations among the events. For conciseness, in diagrams (e.g., in Figure 3.1a), we display only the first conflicting edges (a.k.a. immediate conflicts) with $\sim$ and omit the induced ones. In this event structure $\mathsf{Ld}(X, 0)$ denotes that X reads 0 from the initialization. In addition, $\mathsf{Ld}(X, 1)$ captures the fact that X may read 1 as discussed in § 2.3.1. As a result, the first thread gets two alternative branches in different executions. Similarly in the second thread Y may read 0 or 1 in LB program. However, in Figure 3.1a we show only $\mathsf{Ld}(Y, 1)$ followed by $\mathsf{St}(X, 1)$ execution of the second thread.



(a) One event structure of LB (b) CYC event structure

(c) LB Execution (d) LB Execution (e) CYC Execution

Figure 3.1: One LB event structure and only event structure of CYC. LB event structure has two executions extracted from it, and CYC has only one extracted execution.

3.2 Proposed Approach

We define the semantics of programs in two steps. First, we iteratively construct a "consistent" event structure from the program and then we extract its "consistent" executions.

Event Structure Construction An event structure is constructed by appending one event at a time following the program text. For a write event to be added to the structure, all its po-previous events must have been added; for a read event, there must additionally already exist a justifying write event. Thus, by construction $(\mathsf{po} \cup \mathsf{jf})$ is acyclic. The construction terminates when we can no longer add a new event to the structure. In general, whenever we add an event to an event structure, we check that the resulting structure is *consistent* (i.e., whether it satisfies a few consistency conditions that will be defined later) and discard any event structures that do not satisfy those conditions. Among other things, consistency ensures that sequential programs have their expected semantics by forbidding, for instance, a read to be justified by a stale write—one overwritten by a more recent write to the same location po-before the read.

Event Structures as a Solution to the OOTA Problem in CYC Program Following this approach, we can construct one consistent event structure as shown in Figure 3.1a of LB program. Note that we cannot construct this event structure for program CYC, because once we add the $\mathsf{Ld}(X,0)$ and $\mathsf{Ld}(Y,0)$ events as shown in Figure 3.1e, no further events can be added. Thus these two event structures appropriately distinguish the CYC and LB programs and address the limitation of 'per-execution' based model as discussed in §2.6.

Extracting Execution Next, given a consistent event structure, we extract the set of its maximal conflict-free subsets, which we call *executions*. (These are called configurations by Winskel [75].) As an example (ignoring the rf edges for the moment), Figures 3.1c and 3.1d show the two executions that can be extracted from the event structure in Figure 3.1a. We filter the set of extracted executions by a certain execution consistency predicate, discarding those executions that are inconsistent. The allowed outcomes of a program are determined by these consistent executions.

One standard consistency requirement for executions is that every read should get its value from some write to the same location. This is formalized by a *reads-from* (rf) edge from the write to the read. Other consistency axioms may further constrain the allowed reads-from edges.

In our work, we take execution consistency predicate to be that of RC11 (i.e., the repaired definition of C11 by Lahav et al. [37]) without its $\mathsf{po} \cup \mathsf{rf}$ acyclicity requirement. We thus get a model that is *stronger* than C11 (which allows OOTA) and *weaker* than RC11 (which forbids load-store reordering). In particular, it assigns the right meaning to CYC, LB, and LBfd by restricting $a = b = 1$ outcome in the CYC program and allowing the same outcome in the LB and LBfd programs.

In general, due to different ways in which events may be appended to an event structure, a program may have multiple *consistent* event structures, and each such event structure may yield one or more consistent executions.

3.2.1 A Problem with the Simple Construction Scheme and How to Solve it

Although the simple scheme presented so far works well for LB and its variants, it suffers from a subtle problem demonstrated by the following "random number generator" litmus test.

$$Y = X + 1; \;\Big\|\; X = Y; \;\Big\|\; \begin{array}{l}\textbf{if}\,(X == 100)\\ \quad X = 99;\end{array} \qquad \text{(RNG)}$$

First, as shown in Figure 3.2, even for this obviously terminating program, the event structure construction can go on for ever. Thread 1 reads $X = 0$ and writes $Y = 1$; thread 2 then reads $Y = 1$ and writes $X = 1$. Then, in a conflicting execution branch, thread 1 reads $X = 1$ and writes $Y = 2$; thread 2 reads this value and writes $X = 2$. Then, thread 1 can read $X = 2$ and the process can be repeated indefinitely. This problem is not so serious because one can always stop the construction at some arbitrary point. The more serious problem is that the construction can generate an event structure containing the $\mathsf{St}(X, 99)$ event of the last thread. From there, we can extract the execution shown at the right part of Figure 3.2, which is consistent according to C11 (as witnessed by the depicted rf edges) and leads to a very counterintuitive outcome.

Clearly, the straightforward approach of constructing an event structure does not work here. In this example, we might blame the addition of $\mathsf{Ld}(X, 1)$, as it is the first event that will never appear in any reasonable run of the program. From then on all other events added (shown in

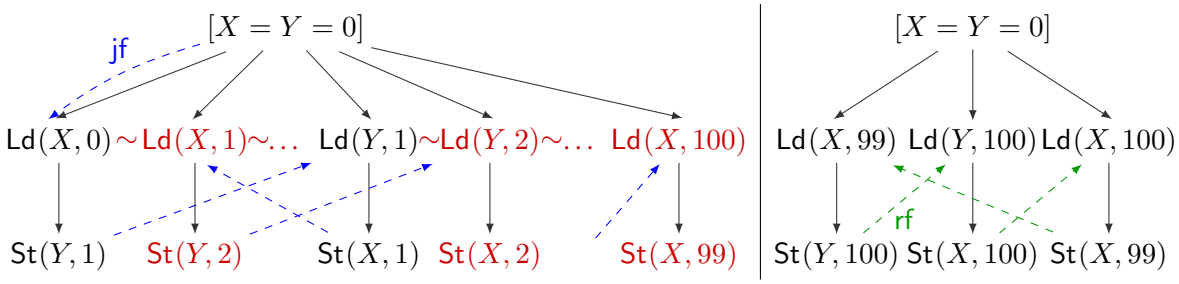


Figure 3.2: A naively constructed event structure of RNG and C11-consistent extracted execution from it.

red in Figure 3.2) are equally bogus. We may observe that $\mathsf{Ld}(X, 1)$ is $(\mathsf{po} \cup \mathsf{jf})^+$-reachable from $\mathsf{Ld}(X, 0)$ in the event structure and thus the creation of $\mathsf{Ld}(X, 1)$ *causally depends* on its conflicting event $\mathsf{Ld}(X, 0)$. As the two events are in conflict, they cannot both occur in a single execution, so perhaps a possible constraint is that *no event should causally depend on a conflicting event.*

Unfortunately, this restriction turns out to be too strong because it also rules out the $a = b = 1$ outcome of LB (cf. Figure 3.1a). We therefore relax the restriction by introducing the notion of *visibility*, and adapt the extraction of executions from event structures to *discard any executions containing invisible events*. As we will shortly see, all events of Figure 3.2 drawn in red are invisible, and so the problematic extracted execution shown to right of the figure is discarded.

To define the notion of visible events, we introduce the *equal-write* (ew) relation, which relates two writes on the same location with the same written value that are in conflicting execution branches. For instance, in Figure 3.1a, the two $\mathsf{St}(Y, 1)$ events are ew-related, whereas in Figure 3.2 no events are ew-related.

Given this relation, we say that an event e is *invisible* whenever there is a conflicting write event in its $(\mathsf{po} \cup \mathsf{jf})^+$-prefix that does not have an equal write in the same execution branch as e (i.e., $(\mathsf{po}^? \cup \mathsf{po}^{-1})$-related to e). For example, the $\mathsf{Ld}(X, 1)$ event is invisible in Figure 3.2 because of the $\mathsf{St}(Y, 1)$ store, whereas $\mathsf{Ld}(X, 1)$ is visible in Figure 3.1a. Note that by definition if an event is invisible in an event structure, then so are all its po-later events.

The equal-write relation also allows to make a formal connection between the *justification* relation (jf) at the level of event structures and the *reads-from* relation (rf) at the level of extracted executions. The idea is to also define rf at the level of event structures in terms of jf and ew. We say that a read event r *reads from* a write w if it is justified by w or by one of its equal writes. When extracting an execution from an event structure, we take their rf relations to match for the extracted events. (We invite the reader to check that this is indeed so for the executions in Figure 3.1).

3.2.2 Coherence in WEAKEST Model

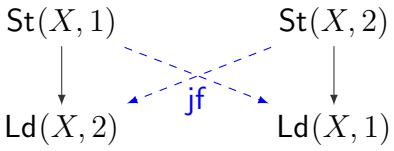
We move on to *coherence*, a basic property enforced by almost all memory models.[1] Coherence states that programs containing only one shared variable exhibit only SC behaviors. To ensure this property, axiomatic memory models require the existence of a (strict) total order among all writes to a given location in an execution. We call the union of all these per-location orders the *modification order* (mo).

With our event structure model, a natural question arises: should we enforce coherence at the level of event structures or only at the level of extracted executions? In this chapter we discuss WEAKEST model that checks coherence only at the execution level. To make this concrete, consider the following standard coherence litmus test program, whose annotated outcome should be forbidden.

$$\begin{array}{l} X = 1; \\ a = X; \ // \, 2 \end{array} \quad \Big\| \quad \begin{array}{l} X = 2; \\ b = X; \ // \, 1 \end{array} \tag{Coh}$$

At the execution level, the outcome $a = 2 \wedge b = 1$ is forbidden because mo must order two writes to X. Suppose without loss of generality that $\mathsf{St}(X, 1)$ is ordered before $\mathsf{St}(X, 2)$. Then, returning $b = 1$ is inconsistent because it reads from a write that is overwritten by a write before the read.

At the event structure level, WEAKEST allows the following event structure:



(Coh-ES)

This, on its own, is not a problem because we cannot extract from it a consistent execution, and so WEAKEST forbids the incoherent outcome.

We move on to a more complex example, consider the program shown in Figure 3.3 and the outcome $a = 3 \wedge b = 2 \wedge c = 1$. We note that this outcome is rather dubious: one can see it as arising due to a violation of coherence or a circular dependency. Indeed, if $b = 2$ then $X = 1$ must be mo-before $X = 2$, and so the first thread cannot read $a = 1$. Likewise, it should not be able to read $a = 3$ because that depends on $c = Y$ reading 1, which circularly depends on the first thread reading $a = 3$. We further note that this outcome is not observable on any machine and the program cannot be transformed in any reasonable fashion so as to enable that outcome. (In particular, read-after-write elimination ruins the outcome, and the only reordering possible is moving the $c = Y$ instruction earlier, which does not enable any further optimizations.)

This outcome is, however, allowed by the WEAKEST model because of the event structure shown in Figure 3.3.

As we will shortly see, this outcome is also allowed by the promising semantics (PS) [33]. In fact, we will show that WEAKEST can indeed simulate PS, which allows us to leverage the existing results about PS.

[1] The exceptions are Itanium and Java, which enforce a slightly weaker coherence property for their plain accesses.

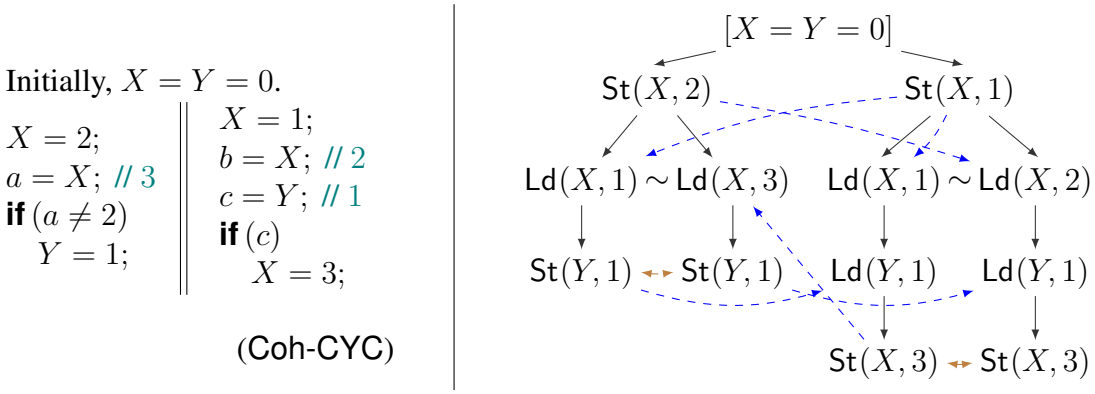


Figure 3.3: A program and its WEAKEST event structure yielding the outcome $a = 3 \wedge b = 2 \wedge c = 1$.

3.3 Formalization: The WEAKEST Model

In this section, we formally define the WEAKEST model that were introduced in the previous section.

We take programs, $\mathbb{P}$, to be top-level parallel compositions of some number of threads, $T_1 \| \ldots \| T_N$. To make our semantics parametric over the programming language syntax, we represent each thread T_i as some non-empty set of traces denoting the possible sequences of memory accesses it can produce. Formally, a trace, τ, is a (finite) sequence of labels given by the following grammar:

$$
\begin{aligned}
lab &::= \mathsf{Ld}_{o_\mathsf{r}}(x, v) \mid \mathsf{St}_{o_\mathsf{w}}(x, v) \mid \mathsf{U}_{o_\mathsf{u}}(x, v, v') \mid \mathsf{F}_{o_\mathsf{u}} && \text{(Event labels)} \\
o_\mathsf{r} &::= \text{NA} \mid \text{RLX} \mid \text{ACQ} \mid \text{SC} && \text{(Memory orders for reads)} \\
o_\mathsf{w} &::= \text{NA} \mid \text{RLX} \mid \text{REL} \mid \text{SC} && \text{(Memory orders for writes)} \\
o_\mathsf{u} &::= \text{RLX} \mid \text{ACQ} \mid \text{REL} \mid \text{ACQ-REL} \mid \text{SC} && \text{(Memory orders for updates and fences)}
\end{aligned}
$$

We have load (Ld), store (St), update (U), and fence (F) labels. Updates are used to model the effect of atomic read-modify-write (RMW) instructions, such as fetch-and-add and compare-and-swap (CAS). The labels record the location accessed (x), the values read and/or written (v, v'), and the corresponding C11 *memory order* (o_r/o_w/o_u): non-atomic (NA), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (ACQ-REL), or sequentially consistent (SC). In increasing strength, these orders are: $\text{NA} \sqsubset \text{RLX} \sqsubset \{\text{ACQ}, \text{REL}\} \sqsubset \text{ACQ-REL} \sqsubset \text{SC}$.

We assume that the thread semantics is *prefix-closed*, *receptive*, and *deterministic*. Prefix-closedness requires for every $\tau \cdot \tau' \in T_i$ (where $\cdot$ denotes trace concatenation), we have $\tau \in T_i$. In particular, this means that the empty trace is always included in T_i. Receptiveness requires that whenever $\tau \cdot lab \in T_i$ and lab is a read label (a load or an update) of location x, then for every value v, there exists a read label lab' of location x reading that value v such that $\tau \cdot lab' \in T_i$. In other words, whenever a thread can read x, it can do so for any possible value. Determinism requires that whenever $\tau \cdot lab \in T_i$ and $\tau \cdot lab' \in T_i$, then the two labels agree except perhaps due to receptiveness. That is, if unequal, then they are both reads (loads or updates) of the same location.

Definition 1. An *event* is a tuple, $\langle id, tid, lab\rangle$, where $id \in \mathbb{N}$ is a unique identifier for the event, $tid \in \mathbb{N}$ identifies the thread to which the event belongs (we use $tid = 0$ for initialization events), and lab denotes the label of the event.

A *read event* is either a load or an update, whereas a *write event* is either a store or an update. Let $\mathcal{E}$ denote the set of all events, $\mathcal{R}$ the set of all read events, $\mathcal{W}$ the set of all write events, and $\mathcal{F}$ the set of fence events. We use subscripts to constraint those sets; e.g., $\mathcal{E}_{\sqsupseteq \text{REL}}$ denotes all events whose memory order is at least REL. We write $e.\mathsf{id}$, $e.\mathsf{tid}$, $e.\mathsf{lab}$, $e.\mathsf{op}$, $e.\mathsf{loc}$, $e.\mathsf{ord}$, $e.\mathsf{rval}$, and $e.\mathsf{wval}$, to return (when applicable) the identifier, thread, label, type ($\mathsf{Ld}, \mathsf{St}, \mathsf{U}, \mathsf{F}$), location, memory order, read value, and written value respectively.

Relational Notation Let $R, S \subseteq \mathcal{E} \times \mathcal{E}$ be binary relations on events. We write $R;S$ for the relational composition of R and S. Let R^{-1} denote the inverse of R, $\mathsf{dom}(R)$ its domain and $\mathsf{codom}(R)$ its range. We write $R^?$, R^+, R^*, $R^=$ for the reflexive, the transitive, the reflexive-transitive, and the reflexive-symmetric closures of R respectively. (Reflexive closure is with respect to the set $\mathcal{E}$, while reflexive-symmetric closure means $R^= \triangleq (R \cup R^{-1})^?$.) Further, $R|_{\mathsf{loc}} \triangleq \{(e, e') \in R \mid e.\mathsf{loc} = e'.\mathsf{loc}\}$ restricts R on pairs of events of the same location. Similarly, $R|_{\neq \mathsf{loc}} \triangleq R \setminus R|_{\mathsf{loc}}$ restricts R on pairs of events of different locations.

The $[A]$ notation denotes an identity relation on set A, i.e. $[A](x, y) \triangleq x = y \wedge x \in A$. For a finite set A that is totally ordered by R, we write $\mathsf{sequence}_R(A)$ to denote the sequence of its elements ordered according to R.

Definition 2. An *event structure*, $G \triangleq \langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}\rangle$, contains the following components:

- $\mathsf{E} \subseteq \mathcal{E}$ is the *set of events* of the event structure containing the set of initialization events, E_0, which has exactly one event with label $\mathsf{St}_{\text{NA}}(x, 0)$ for every location x. For a thread i, we write E_i to refer to the events of the event structure with thread identifier i.

- $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ denotes the *program order*. It is a strict partial order on events recording when an event precedes another one in the control flow of the program. Initialization events are po-before all non-initialization events (i.e., $\mathsf{E}_0 \times (\mathsf{E} \setminus \mathsf{E}_0) \subseteq \mathsf{po}$), and that non-initialization events related by po have the same thread identifier.

  Further, we assume that non-initialization po-predecessors of an event are totally ordered by po. That is, we never have distinct $a, b \in \mathsf{E} \setminus \mathsf{E}_0$ that have a common po-successor and are not po-related to one another. In relational notation, $[\mathsf{E} \setminus \mathsf{E}_0] ; \mathsf{po} ; \mathsf{po}^{-1} ; [\mathsf{E} \setminus \mathsf{E}_0] \subseteq \mathsf{po}^=$.

  We note that po is not total over events of the same thread. In particular, we say that events of the same thread not related by the program order are in *conflict* with one another.

$$G.\mathsf{cf} \triangleq \{(a, b) \in G.\mathsf{E} \times G.\mathsf{E} \mid a.\mathsf{tid} = b.\mathsf{tid} \neq 0\} \setminus \mathsf{po}^= \qquad \text{(Conflict relation)}$$

  By definition, the conflict relation, cf, is symmetric and irreflexive. Moreover, it is forward-closed with respect to the program order (i.e., $\mathsf{cf} ; \mathsf{po} \subseteq \mathsf{cf}$): if two events conflict, then so are all their future events.

- $\mathsf{jf} \subseteq \mathsf{E} \times \mathsf{E}$ is the *justified-from* relation, which relates a write event to the reads it justifies. That is, whenever $\mathsf{jf}(w, r)$, then $w \in \mathcal{W}$, $r \in \mathcal{R}$, $w.\mathsf{loc} = r.\mathsf{loc}$ and $w.\mathsf{wval} = r.\mathsf{rval}$ hold. We require that every read is justified from exactly one write. That is, jf^{-1} is functional and $\mathsf{dom}(\mathsf{jf}) = \mathsf{E} \cap \mathcal{R}$.

- $\mathsf{ew} \subseteq [\mathcal{W}_{\sqsubseteq \mathrm{RLX}}] ; \mathsf{cf}|_{\mathsf{loc}} ; [\mathcal{W}_{\sqsubseteq \mathrm{RLX}}]$ is the *equal-writes* relation, a symmetric relation relating conflicting (relaxed) writes on the same location writing the same value.

Given an event structure G, we use notation $G.X$ to project the X component out of G. When G is clear from the context, we sometimes omit the "$G.$".

We now define a number of derived relations on event structures.

First, we define *synchronizes-with* (sw) and *happens-before* (hb) following RC11 [37] (replacing C11's reads-from relation with justification). The sw definition is quite involved and states conditions under which a justification edge synchronizes two threads. The simplest case inducing synchronization is when an acquire read event reads from a release write. More advanced cases involve release/acquire fences and/or reads through sequences of updates (known as release sequences in C11). An event a *happens before* an event b if there is a path from a to b consisting of program order and synchronization edges.

$$G.\mathsf{sw} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; G.\mathsf{po})^? ; [\mathcal{W}] ; G.\mathsf{po}|_{\mathsf{loc}}^? ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; G.\mathsf{jf}^+ ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (G.\mathsf{po} ; [\mathcal{F}])^? ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}] \quad \text{(Synchronizes-with)}$$

$$G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup G.\mathsf{sw})^+ \quad \text{(Happens-before)}$$

We say that two events are in *immediate conflict* ($\sim$) if neither of them has a po-previous event conflicting with the other event. Two events are in *extended conflict* (ecf) if they happen after conflicting events, which means that they cannot be part of the same execution.

$$G.\sim \triangleq G.\mathsf{cf} \setminus (G.\mathsf{cf} ; G.\mathsf{po} \cup G.\mathsf{po}^{-1} ; G.\mathsf{cf}) \quad \text{(Immediate conflict)}$$

$$G.\mathsf{ecf} \triangleq (G.\mathsf{hb}^{-1})^? ; G.\mathsf{cf} ; G.\mathsf{hb}^? \quad \text{(Extended conflict)}$$

Next, the *reads-from* relation, rf, lifts the justified-from relation to all equal writes; i.e., whenever r is justified by w and w is equal to w' and w' and r do not conflict, then $\mathsf{rf}(w', r)$. This, in particular, means that rf^{-1} is not necessarily functional at the level of event structures.

$$G.\mathsf{rf} \triangleq (G.\mathsf{ew}^? ; G.\mathsf{jf}) \setminus G.\mathsf{cf} \quad \text{(Reads-from relation)}$$

Since the WEAKEST model does not record the modification order, we define $\mathsf{mo}_{\mathsf{strong}}$, a stronger version of the modification order, which relates writes to the same location that are ordered by happens-before. We then use $\mathsf{mo}_{\mathsf{strong}}$ in place of mo to derive strong read-before relation $\mathsf{fr}_{\mathsf{strong}}$. Finally, the strong extended coherence order is the transitive closure of the union of the three relations (rf, $\mathsf{mo}_{\mathsf{strong}}$, and $\mathsf{fr}_{\mathsf{strong}}$).

$$G.\mathsf{mo}_{\mathsf{strong}} \triangleq [\mathcal{W}] ; G.\mathsf{hb}|_{\mathsf{loc}} ; [\mathcal{W}] \quad \text{(Strong modification order)}$$

$$G.\mathsf{fr}_{\mathsf{strong}} \triangleq (G.\mathsf{rf}^{-1} ; G.\mathsf{mo}_{\mathsf{strong}}) \setminus [\mathcal{E}] \quad \text{(Strong reads-before)}$$

$$G.\mathsf{eco}_{\mathsf{strong}} \triangleq (G.\mathsf{rf} \cup G.\mathsf{mo}_{\mathsf{strong}} \cup G.\mathsf{fr}_{\mathsf{strong}})^+ \quad \text{(Strong extended coherence order)}$$

Remark 1. Note that the definitions above allow ew to relate a relaxed store with a conflicting relaxed update writing the same value to the same location. We could eliminate this flexibility by enforcing that ew relates either a pair of stores or a pair of updates. These restrictions would suffice for ew to relate only events with the same label and, in the case of updates, with the same justification (jf). Under these restrictions, then sw as defined above is equivalent to the C11-definition of sw that uses rf in the place of jf. Without these restrictions, however, the definition with rf is problematic because adding a new update event and making it ew-related to an existing store could induce synchronizations between existing events in the event structure. The only reason why we did not restrict ew is so that we can establish that WEAKEST is weaker than the promising semantics in Chapter 4. The promising semantics, in particular, allows certifying a promised message with an update event and then fulfilling it with a write, and vice versa.

3.3.1 Event Structure Consistency Checking

We say that an event e is visible in an event structure G if all the writes recursively used to justify e that are in conflict with e have some equal write that does not conflict with e. Formally:

Definition 3. An event e is *visible* in an event structure G if $e \in G.\mathsf{E}$ and

$$[\mathcal{W}] \,;\, (G.\mathsf{cf} \cap G.\mathsf{jfe}; (G.\mathsf{po} \cup G.\mathsf{jf})^*; G.\mathsf{jfe}; G.\mathsf{po}^?) \,;\, [\{e\}] \subseteq G.\mathsf{ew} \,;\, G.\mathsf{po}^=$$

where $G.\mathsf{jfe} \triangleq G.\mathsf{jf} \setminus G.\mathsf{po}$ denotes the *external justification edges*. We write $\mathsf{vis}(G)$ for the set of all visible events of an event structure G.

As we will shortly see, our model essentially treats extended conflicts as normal conflicts.

Definition 4. An event structure G is *consistent* according to the WEAKEST model, written $\mathsf{isCons}_{\text{WEAKEST}}(G)$, iff the following conditions hold:

- (CF) No event can be in extended conflict with itself: $G.\mathsf{ecf}$ is irreflexive.
- (CFJ) A write cannot justify a read in extended conflict: $G.\mathsf{jf} \cap G.\mathsf{ecf} = \emptyset$.
- (VISJ) Only visible events justify reads of other threads: $\mathsf{dom}(G.\mathsf{jfe}) \subseteq \mathsf{vis}(G)$.
- (ICF) Immediately conflicting events must be reads: $G.\sim \subseteq \mathcal{R} \times \mathcal{R}$.
- (ICFJ) Immediately conflicting reads cannot be justified by the same or by equal writes: $G.\mathsf{jf} \,;\, G.\sim \,;\, G.\mathsf{jf}^{-1} \,;\, G.\mathsf{ew}^?$ is irreflexive.
- (COH) $G.\mathsf{hb} \,;\, G.\mathsf{eco}^?_{\mathsf{strong}}$ is irreflexive.

We now explain these constraints in order.

The first constraint (CF) ensures that every event in the event structure could belong to some execution by checking that its hb-predecessors are conflict-free. This, in particular, ensures that po-related reads do not observe values from two different execution branches of

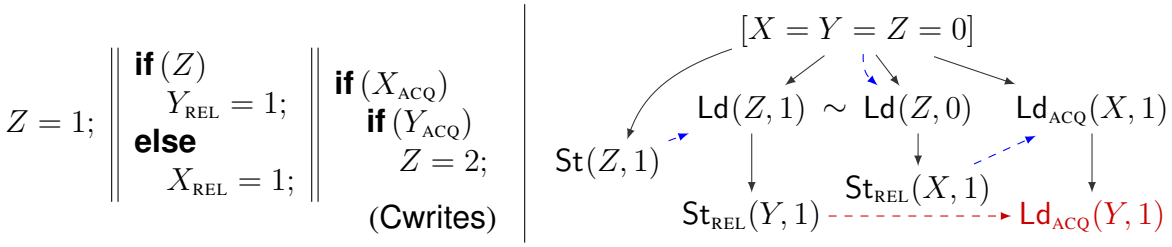


Figure 3.4: Constraint (CF) restricts $Z = 2$ in the Cwrites program.

some other thread in the case where both reads-from edges would result in a synchronization. We explain the scenario with the Cwrites program in Figure 3.4 taken from Chakraborty and Vafeiadis [22]. Intuitively, this program should never produce the outcome $Z = 2$, as there is no execution under which both X and Y could have the value 1. With this in mind, we would like to rule out the event structure shown in Figure 3.4 that contains an execution branch where both acquire loads of X and Y read the value 1. It is easy to see that the (CF) does not hold in the event structure, as $\mathsf{St}_{\text{REL}}(Y, 1)$ conflicts with $\mathsf{St}_{\text{REL}}(X, 1)$ which happens before $\mathsf{Ld}_{\text{ACQ}}(Y, 1)$.

The second constraint (CFJ) forbids a read to be justified from a write event in extended conflict. Intuitively, since conflicting events can never appear in a single execution, allowing reads to read from conflicting events could generate OOTA results. This constraint rules out the problematic event structure for the variants of Cwrites where one of the reads of the third thread is relaxed.

Remark 2. While (CF) rules out the problematic event structure of Figure 3.4, it is not strictly needed for rule out the $Z = 2$ behavior, because no RC11-consistent execution can be extracted from that event structure. Nevertheless, (CF) and (CFJ) are needed for a more complicated variant of Cwrites in Kang et al. [33, Appendix A.2], where without them we cannot guarantee DRF-RA (Theorem 4).

The third constraint (VISJ) says that reads are either justified locally from the same thread or they are justified from some *visible* write. This constraint forbids the undesired event structure in Figure 3.2 for the RNG program.

The next two constraints place some restrictions on immediate conflicts that rule out creating unnecessary duplication in the event structure. (ICF) requires that immediate conflicts are between read events. Because we assume that the thread semantics is deterministic, immediately conflicting events must be created by the same program instruction. If, for example, they originate from a load instruction, they will all be load events. If they originate for a compare-and-swap (CAS), they will either be update events (for the successful case) or load events (for the case when the CAS fails). In both of these cases, it may well make sense to have an immediate conflict in the event structure. In contrast, it does not make sense to create an immediate conflict from a store or a fence instruction, because both events will have the exact same label, which is why (ICF) rules out such events. Similarly, (ICFJ) disallows creating an immediate conflict by reading from the same write. That is, we cannot have two reads in immediate conflict that are justified by the same write or by equal writes. Again such reads will read the same value, and so will lead to duplication in the event structure.

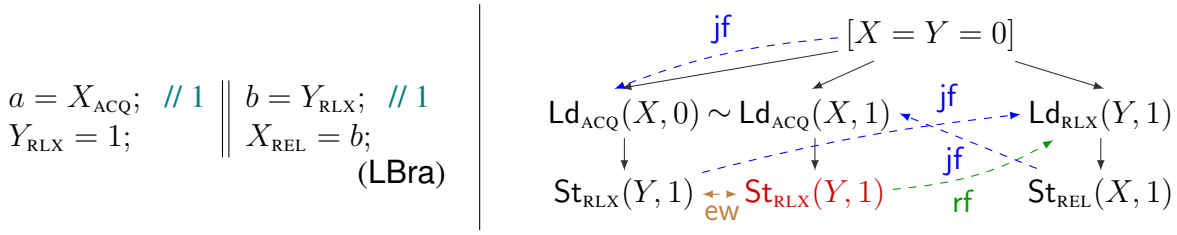


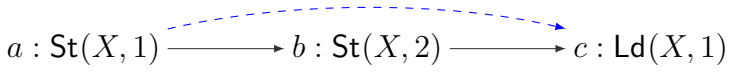
Figure 3.5: Outcome $a = b = 1$ is not allowed in LBra because of a coherence violation.

$$\frac{\begin{array}{c} A \subseteq G.\mathsf{E}_{e.\mathsf{tid}} \quad \mathsf{dom}([\mathsf{E}_{e.\mathsf{tid}}]\,;\mathsf{po}\,;[A]) \subseteq A \quad \mathsf{labels}(\mathsf{sequence}_{\mathsf{po}}(A)) \cdot e.\mathsf{lab} \in \mathbb{P}(e.\mathsf{tid}) \\ \mathsf{E}' = \mathsf{E} \uplus \{e\} \quad \mathsf{po}' = \mathsf{po} \cup (A \times \{e\}) \quad \mathsf{isCons}_M(\langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle) \quad CF = (\mathsf{E}_{e.\mathsf{tid}} \setminus A) \\ \textbf{if } e \in \mathcal{R} \textbf{ then } \exists w \in \mathsf{E} \cap \mathcal{W}.\ \mathsf{jf}' = \mathsf{jf} \cup \{(w, e)\} \wedge w.\mathsf{loc}{=}e.\mathsf{loc} \wedge w.\mathsf{wval}{=}e.\mathsf{rval} \textbf{ else } \mathsf{jf}'{=}\mathsf{jf} \\ \textbf{if } e \in \mathcal{W}_{\sqsubseteq \mathsf{RLX}} \textbf{ then } \mathsf{AddEW}(\mathsf{ew}, \mathsf{ew}', CF, e) \textbf{ else } \mathsf{ew}' = \mathsf{ew} \\ \textbf{if } e \in \mathcal{W} \textbf{ then } \mathsf{AddMO}(\mathsf{mo}, \mathsf{mo}', \mathsf{E}, CF, \mathsf{ew}, e) \textbf{ else } \mathsf{mo}' = \mathsf{mo} \end{array}}{\langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo} \rangle \rightarrow_{\mathbb{P},M} \langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle}$$

Figure 3.6: One construction step of a program's event structure, where $\mathsf{labels}(a_1 \cdots a_n) \triangleq a_1.\mathsf{lab} \cdots a_n.\mathsf{lab}$.

Finally, the (COH) constraint enforces that the event structure is coherent at various levels. (COH) constraint ensure that the happens-before (hb) is irreflexive and that rf does not contradict hb. We explain the constraint with the program in Figure 3.5, which is another variant of the load buffering litmus test. In this case, the outcome $a = b = 1$ must be forbidden and is actually disallowed by C/C++11 (despite thin-air problems). The reason is that if $a = 1$, then the release write of X synchronizes with the acquire read of X, and so the read of Y would have to read from an hb-later write. The coherence constraint (COH) similarly enforces that the event structure in Figure 3.5 is inconsistent. More specifically, the event structure without the second $\text{St}_{\text{RLX}}(Y, 1)$ event (i.e., the one displayed in red) is consistent but adding that event along with the displayed ew relation renders the event structure inconsistent. The equal-writes relation together with Y's justification edge induce the displayed reads-from edge, which contradicts happens-before.

In addition, the coherence constraints also ensure that reads do not read overwritten values, as for example in the following inconsistent event structure:



(Basic coherence violation)

The justification edge induces an rf-edge from a to c, and in turn an $\mathsf{fr}_{\mathsf{strong}}$-edge from c to b, which contradicts hb.

3.3.2 Event Structure Construction

Having defined event structures and their consistency, we move on to explain how they are constructed.

We start with the initial event structure G_{init} that contains as events only the initialization events, $\mathsf{E}_0 = \{a_1, \ldots, a_k\}$ where k is the number of variables in the program and each a_i has label $\mathsf{St}_{\text{NA}}(X_i, 0)$ and thread identifier 0. All the other components of G_{init} are simply the empty relation.

Starting with G_{init}, we construct an event structure of a program $\mathbb{P}$ incrementally by adding one event at a time using the rule shown in Figure 3.6. The reduction relation has the form $G \rightarrow_{\mathbb{P},\text{WEAKEST}} G'$ and represents a transition from event structure G to event structure G' of the program $\mathbb{P}$ according to WEAKEST model.

The rule adds one new event e to the event structure G. To do so, it picks the set A of its po-predecessors from the same thread. This set A includes events of thread $e.\mathsf{tid}$ that are already in the event structure ($\mathsf{E}_{e.\mathsf{tid}}$), and is closed under program order prefix. That is, any po-predecessor of an event in A should also belong to A. Next, it checks that local thread semantics can produce the label of e, after the sequence of labels of A.

In the second line of the rule, the set of events is extended to include e, and the program order is similarly extended to place e after all the events in A. It checks that the resulting event structure is consistent according to WEAKEST model where the updates to the event structure's remaining components will be defined shortly, and discards any inconsistent event structures. We finally let CF contain all the events conflicting with e, which are exactly those events from the same thread as e that were not placed before it.

The third line of the rule concerns the update of the justification relation. If e is a read event, then there must already be some write event w in the event structure that writes the value that e reads to the same location. In that case, the justification relation is extended with the edge (w, e). If e is not a read (i.e., it is a store or a fence event), then jf remains unchanged.

The last line concerns updates the equal-writes relation whenever the new event is a relaxed write. We use the following auxiliary definition for updating ew:

$$\mathsf{AddEW}(\mathsf{ew}, \mathsf{ew}', CF, e) \triangleq \exists W \subseteq \mathcal{W}_{\sqsubseteq \text{RLX}} \cap CF.\ (\forall w \in W.\ w.\mathsf{loc} = e.\mathsf{loc} \wedge w.\mathsf{wval} = e.\mathsf{wval}) \\ \wedge\ \mathsf{ew}' = \mathsf{ew} \cup (W \times \{e\}) \cup (\{e\} \times W)$$

AddEW selects a set W of conflicting relaxed writes to the same location and that write the same value as e. Then, ew is extended by relating e to all the elements in W.

3.3.3 Execution Extraction in the WEAKEST Model

As mentioned in §3.2, after a consistent event structure is constructed, we extract from it a set of conflict-free subsets of events, which we call executions.

More formally, an execution X is a tuple of the form $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ whose components in order denote the set of events, the program order, the reads-from relation, and the modification order of the execution. As with event structures, we use the dot notation (e.g., X.E) to project the relevant components of an execution.

The set of events, the program order, and the reads-from relation have the same constraints as for event structures. For example, when event r reads from event w, then r is a read, w is a write, they both access the same location, and the value written by w is read by r. Unlike event structures, however, executions do have the equal write (ew) relation. (Since an execution has

$$\mathsf{X.sw_{C11}} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; \mathsf{X.po})^{?} ; [\mathcal{W}] ; \mathsf{X.po}|^{?}_{\mathsf{loc}} ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; \mathsf{X.rf}^{+} ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (\mathsf{X.po} ; [\mathcal{F}])^{?} ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}]$$

$$\mathsf{X.hb_{C11}} \triangleq (\mathsf{X.po} \cup \mathsf{X.sw_{C11}})^{+}$$

$$\mathsf{X.scb} \triangleq \mathsf{X.po} \cup (\mathsf{X.po}|_{\neq \mathsf{loc}} ; \mathsf{X.hb_{C11}} ; \mathsf{X.po}|_{\neq \mathsf{loc}}) \cup \mathsf{X.hb_{C11}}|_{\mathsf{loc}} \cup \mathsf{X.mo} \cup \mathsf{X.fr}$$

$$\mathsf{X.psc_{base}} \triangleq [\mathcal{E}_{\mathrm{SC}}] ; ([\mathcal{F}_{\mathrm{SC}}] ; \mathsf{X.hb_{C11}})^{?} ; \mathsf{X.scb} ; (\mathsf{X.hb_{C11}} ; [\mathcal{F}_{\mathrm{SC}}])^{?} ; [\mathcal{E}_{\mathrm{SC}}]$$

$$\mathsf{X.psc_{F}} \triangleq [\mathcal{F}_{\mathrm{SC}}] ; (\mathsf{X.hb_{C11}} \cup \mathsf{X.hb_{C11}} ; \mathsf{X.eco} ; \mathsf{X.hb_{C11}}) ; [\mathcal{F}_{\mathrm{SC}}]$$

Figure 3.7: Additional definitions needed for execution consistency (taken from [37]).

no conflicting events, then ew would necessarily be empty.) Similarly, the modification order of executions (X.mo) is total on writes to the same location.

In order to define execution consistency, we need some additional definitions, which are taken straight from Lahav et al. [37] and are presented in Figure 3.7. Here, *synchronizes-with* is defined as in C11 referring to the reads-from relation rather than jf, which does not exist at the level of executions. The last three definitions (scb, $\mathsf{psc_{base}}$, and $\mathsf{psc_F}$) are quite technical and concern cases when an ordering between SC events is guaranteed to be preserved by the memory model. They are not important for our examples, and are only included here for completeness.

Next, we define when an execution is consistent. As mentioned, we take the RC11 consistency constraints [37], except that we allow $(\mathsf{X.po} \cup \mathsf{X.rf})^{+}$-cycles. The reason for allowing cycles is that we do not rely on RC11 consistency to rule out OOTA behaviors, but rather on the construction of event structures. One further (minor) difference is that we model updates as single events, whereas Lahav et al. [37] model them as two events (a read and a write) connected by a special read-modify-write relation. As a result, our atomicity condition is slightly different.

Definition 5. An execution X is *consistent*, denoted $\mathsf{isCons}(\mathsf{X})$, if the following hold:

(Total-RF) $\mathsf{codom}(\mathsf{X.rf}) = \mathsf{X.E} \cap \mathcal{R}$;

(Total-MO) $\mathsf{X.mo}^{=} = ((\mathsf{X.E} \cap \mathcal{W}) \times (\mathsf{X.E} \cap \mathcal{W}))|_{\mathsf{loc}}$;

(Coherence) $\mathsf{X.hb_{C11}} ; \mathsf{X.eco}^{?}$ is irreflexive;

(Atomicity) X.fr ; X.mo is irreflexive; and

(SC) $\mathsf{X.psc_{base}} \cup \mathsf{X.psc_F}$ is acyclic.

where eco and fr are defined as for event structures.

The consistency constraints require that (1) every read of the execution reads from some write; (2) the modification order, mo, is total over all writes to the same location; (3) the execution is coherent (i.e., the execution's restriction to accesses of only one location is SC); (4) atomic updates read from their immediate mo-predecessor; and (5) certain cycles going through SC atomics are disallowed. Among other cases, it ensures that (1) putting SC fences

between every two accesses in each thread guarantees SC, and (2) if all accesses of a consistent execution are sequentially consistent, then the execution is SC.

We move on to define how an execution is extracted from an event structure. First, we have to select an appropriate set of events, A, from the event structure so that restricting the event structure to those events gives us an execution. We place several constraints: (1) all those events should be visible; (2) they should not conflict with one another; and (3) they should be *prefix-closed* with respect to hb: i.e., every hb-predecessor of an event in A should also be in A; and Formally,

$$\mathsf{GoodRestriction}(G) \triangleq \left\{ A \;\middle|\; A \subseteq \mathsf{vis}(G) \;\wedge\; [A]\,;G.\mathsf{cf}\,;[A] = \emptyset \;\wedge\; \mathsf{dom}(G.\mathsf{hb}\,;[A]) \subseteq A \right\}$$

Second, given such a conflict-free set of events, $A \subseteq G.\mathsf{E}$, we define how to restrict G to get an execution. The following formal definition, $\mathsf{Project}_{\text{WEAKEST}}(G, A)$, does this by simply restricting po, rf, and mo to the set of events, A. Since, however, WEAKEST does not record the modification order in event structures, $\mathsf{Project}_{\text{WEAKEST}}(G, A)$ generates an arbitrary mo.

$$\mathsf{Project}_{\text{WEAKEST}}(G, A) \triangleq \{\langle A\,, G.\mathsf{po} \cap (A \times A)\,, G.\mathsf{rf} \cap (A \times A)\,, mo\rangle \mid mo \subseteq (A \times A)\}$$

Putting these two components together, the set of executions of event structure G are those consistent executions that are a result of projecting G to some good restriction of its events.

$$\mathsf{ex}_{\text{WEAKEST}}(G) \triangleq \{\mathsf{X} \mid \exists A \in \mathsf{GoodRestriction}(G).\ \mathsf{X} \in \mathsf{Project}_{\text{WEAKEST}}(G, A) \wedge \mathsf{isCons}_{\text{WEAKEST}}(\mathsf{X})\}$$

Note that the executions extracted from G are not necessarily *maximal*. Rather, for any execution extracted from G, all its consistent $(\mathsf{po} \cup \mathsf{rf})$-prefixes can also be extracted from G.

3.3.4 Program Behaviors

The final step is to define program behaviors. We define the behavior of an execution to be the final contents of the memory at the end of an execution, i.e., the value written by the mo-maximal write for each location.

$$\mathsf{Behavior}(\mathsf{X}) \triangleq \{(e.\mathsf{loc}, e.\mathsf{wval}) \mid \exists e \in \mathsf{X.E} \cap \mathcal{W}.\ [\{e\}]\,;\mathsf{X}.\mathsf{mo} = \emptyset\}$$

Then, the set of behaviors of a program under WEAKEST model contains the behaviors of any *maximal* execution X that can be extracted from an event structure G that was constructed from the program by following the model.

$$\mathsf{Behavior}_{\text{WEAKEST}}(\mathbb{P}) \triangleq \{\mathsf{Behavior}(\mathsf{X}) \mid \exists G.\ \ G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKEST}}{}^{*}\ G \ \wedge\ \mathsf{X} \in \mathsf{ex}_{\text{WEAKEST}}(G) \wedge\ \mathsf{maximal}_{\mathbb{P}}(\mathsf{X})\}$$

where $\mathsf{maximal}_{\mathbb{P}}(\mathsf{X}) \triangleq \nexists i, lab.\ \mathsf{labels}(\mathsf{sequence}_{\mathsf{X.po}}(\mathsf{X.E}_i)) \cdot lab \in \mathbb{P}(i)$. Maximality ensures that all threads have terminated according to the thread semantics.

Chapter Summary In this chapter we have discussed our proposed WEAKEST model based on event structure. We demonstrate WEAKEST event structure construction along with execution extraction from a constructed WEAKEST event structure. In the next chapter, we relate WEAKEST model to the promising semantics [33].

4 The WEAKEST Model and Promising Semantics

In this chapter, we relate the WEAKEST model to the promising semantics (PS) of Kang et al. [33]. We start with an example where given an execution in PS we construct an WEAKEST with an execution having the same behavior as the PS execution. Next, we introduce the details of PS and then, in §4.3, we show that WEAKEST is weaker than PS.

4.1 Relating WEAKEST to the Promising Semantics

We start with a simplified informal presentation of promising semantics sufficient for programs containing only relaxed loads and stores. (We refer the reader to Kang et al. [33] for further details.) PS is defined operationally by a machine, whose state consists of the states of the threads and the shared memory. The shared *memory* is modeled as a set of messages, representing the writes that the program has performed. Messages are of the form $\langle x : v@t \rangle$ denoting a write of value v to location x at timestamp t.

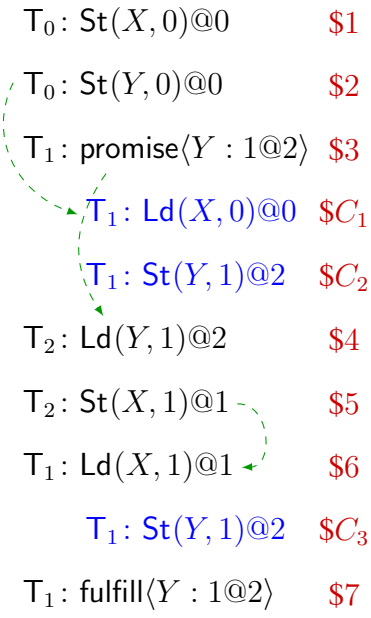
Timestamp. In promising semantics each write operation is tagged with a timestamp. The timestamps on a location enforce a total order and in an execution the timestamps on a location are dense, that is, given two adjacent timestamps on a location, we can always assign an intermediate timestamp to a write on the same location. The timestamps on a location decide the order of the write operations in an execution in the promising semantics. Based on the timestamps on each location, promising semantics introduce the concept of *view* for each thread.

View. Each thread maintains a *view*, V, mapping each memory location to a timestamp, indicating the message with the maximum timestamp that the thread is aware of. When a thread performs a store $\mathsf{St}(x, v)$, it chooses an unused timestamp $t > V(x)$, adds the message $\langle x : v@t \rangle$ to memory, and updates its thread view of x to timestamp t. When it performs a load of x, it reads from a message $\langle x : v@t \rangle$ with $t \geq V(x)$ and updates its thread view of x to timestamp t.

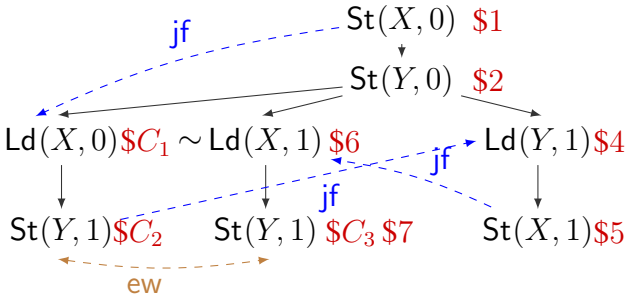
Promising semantics uses timestamps and views to enforce coherence. Consider the Coh program from §3.2.2:

$$\begin{array}{l|l} X = 1; & X = 2; \\ a = X; \ // \ 2 & b = X; \ // \ 1 \end{array}$$

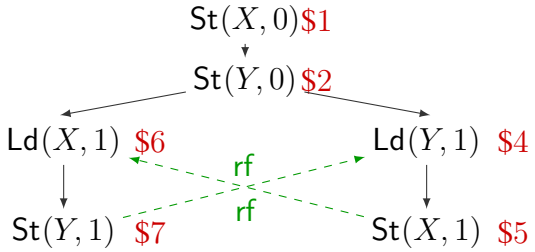
Coherence disallows $a = 2 \wedge b = 1$ outcome in any execution. Now we explain how promising semantics enforce coherence. Consider X is initialized at timestamp $@0$ and hence the initial views of each thread is $\langle X@0 \rangle$. Let $X = 1$ takes place at timestamp $@1$, and hence the view of the first thread is changed to $\langle X@1 \rangle$. Similarly the second thread updates its view to $\langle X@2 \rangle$



(a) Promise steps of LB.



(b) The corresponding event structure in the WEAKEST model.



(c) The corresponding C11-consistent extracted execution.

Figure 4.1: A promise machine execution of LB with its corresponding event structure and extracted execution.

when write $X = 2$ is performed at timestamp @2. When $a = X$ reads from $X = 2$, then the view of first thread is updated to $\langle X@2 \rangle$. The second thread already has a view $\langle X@2 \rangle$ and hence reading from $X : 1@1$ results in an older view $\langle X@1 \rangle$ in the second thread. As a result $b = 1$ is not possible.

Thus timestamps and views enforce coherence in promising semantics. In addition, promising semantics enforce some more primitives which we discuss now.

Promise and Fulfill. Besides executing its next instruction, a thread can also *promise* to write value v at some location x at some future timestamp $t > V(x)$. It does so by appending the message $\langle x : v@t \rangle$ to memory as if the write were actually performed, and keeping track of the message as being an outstanding promise in its local state. To make such a promise, the respective thread must be able to *certify* its promise: namely, to be able to perform a write fulfilling the promise in a finite number of thread-local steps. More generally, PS requires that after each execution step all outstanding promises be certifiable. When a write step later fulfills an outstanding promise, it marks it as fulfilled.

Example 1 (Load buffering)**.** Figure 4.1a displays an execution of the promise machine for LB resulting in the outcome $a = b = 1$. The steps are as follows.
($1,$2) Initially, X and Y are zero and the memory is $\{\langle X : 0@0 \rangle, \langle Y : 0@0 \rangle\}$.
($3) Thread T_1 promises to write $Y = 1$ at timestamp 2 and appends a message $\langle Y : 1@2 \rangle$ to the memory. The certification steps read from message $\langle X : 0@0 \rangle$ (C_1) and fulfill the promise (C_2).
($4) T_2 executes $b = Y$; and reads from message $\langle Y : 1@2 \rangle$.

($5) T_2 executes $X = 1$; adding message $\langle X : 1@1 \rangle$ to the memory.
($6) T_1 executes $a = X$; reading message $\langle X : 1@1 \rangle$. Note that T_1 can still fulfill its promise by $\$C_3$.
($7) Finally, T_1 executes the $Y = 1$; statement and fulfills its promise. □

Now consider the CYC program:

$$\begin{array}{l} a = X; \\ \mathsf{if}(a) \\ \quad Y = 1; \end{array} \;\Big\|\; \begin{array}{l} b = Y; \\ \mathsf{if}(b) \\ \quad X = 1; \end{array}$$

In this program we cannot promise $X = 1$ in the first thread or $Y = 1$ in the second thread as there is no certificate to fulfill these promises. As a result, CYC program cannot have $a = b = 1$ outcome in promising semantics.

Thus using the concept of *promise* and *certificate*, the promising semantics allows $a = b = 1$ in LB program and disallows $a = b = 1$ in CYC program.

Example of Simulating Promising Semantics by WEAKEST We now demonstrate on the load buffering example how we can simulate the promise machine execution with our WEAKEST model. Figure 4.1b displays the corresponding event structure following the steps of the promise machine execution.

($1, $2) We add the two initialization stores $\mathsf{St}(X, 0)$ and $\mathsf{St}(Y, 0)$ at the beginning of the graph.

($3) For a promise step, we cannot immediately add a corresponding store to the event structure. We therefore look at its promise-free certificate execution (i.e., steps $\$C_1$ and $\$C_2$) and add the corresponding events to the structure. This adds events $\mathsf{Ld}(X, 0)$ and $\mathsf{St}(Y, 1)$.

($4) In the promise machine, this step reads from the promised message $\langle Y : 1@2 \rangle$. Recall that the corresponding event in the event structure (i.e., $\$C_2$) was created by the certificate run. So we append a $\mathsf{Ld}(Y, 1)$ event justifying from $\$C_2$.

($5) We simply append $\mathsf{St}(X, 1)$ to the event structure.

($6) We construct event $\mathsf{Ld}(X, 1)$ justifying from $5 (i.e., the event corresponding to message $\langle X : 1@1 \rangle$). which is appended immediately after $\mathsf{St}(Y, 0)$ in the first thread. As $\mathsf{St}(Y, 0)$ already has an immediate po-following event in the first thread, $\mathsf{Ld}(X, 1)$ is in conflict with $\mathsf{Ld}(X, 0)$.

Note that at this point, the newly added event $6 is *invisible* because of the conflicting store $\$C_2$ that is justified from thread T_2. To make $6 visible, we look at T_1's certification run and add the corresponding events to the event structure (i.e., event $\$C_3$). Since the certificate must fulfill all outstanding promises, it must contain a write producing the same message as that of step $\$C_2$ (namely, the write $\$C_3$). We relate the two writes by ew, which makes events $6 and $\$C_3$ visible.

T_0: $\mathsf{St}(X,0)@0$ $\$1$

T_0: $\mathsf{St}(Y,0)@0$ $\$2$

T_2: $\mathsf{St}(X,1)@2$ $\$3$

T_1: promise$\langle Y:1@5\rangle$ $\$4$

T_1: $\mathsf{St}(X,2)@1$ $\$C_1$

T_1: $\mathsf{Ld}(X,1)@2$ $\$C_2$

T_1: $\mathsf{St}(Y,1)@5$ $\$C_3$

T_2: promise$\langle X:3@10\rangle$ $\$5$

T_2: $\mathsf{Ld}(X,1)@2$ $\$C_4$

T_2: $\mathsf{Ld}(Y,1)@5$ $\$C_5$

T_2: $\mathsf{St}(X,3)@10$ $\$C_6$

T_1: $\mathsf{St}(X,2)@7$ $\$6$

T_2: $\mathsf{Ld}(X,2)@7$ $\$7$

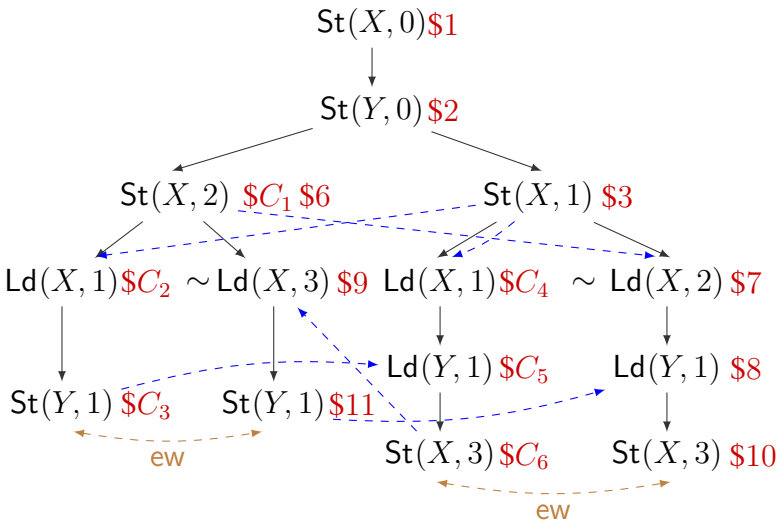
T_2: $\mathsf{Ld}(Y,1)@5$ $\$8$

T_1: $\mathsf{Ld}(X,3)@10$ $\$9$

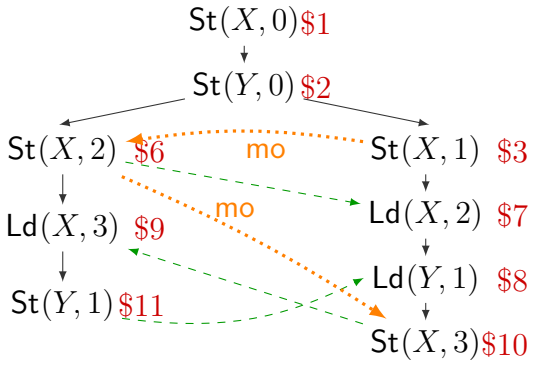
T_2: fulfill$\langle X:3@10\rangle$ $\$10$

T_1: fulfill$\langle Y:1@5\rangle$ $\$11$

(a) Promise execution steps



(b) Constructed WEAKEST event structure



(c) Extracted execution

Figure 4.2: A promise execution of Coh-CYC with its corresponding WEAKEST event structure and execution.

($7) At this step, we would normally append a $\mathsf{St}(Y,1)$ event immediately after event $6, but we notice that such an event already exists because of the previous step. We therefore do nothing and simply record that $7 corresponds to the same event as $\$C_3$.

By selecting the non-promised events, we can then extract the execution shown in Figure 4.1c. The constraints about the messages' timestamps ensure that the execution is consistent.

Example 2 (Coh-CYC from Figure 3.3)**.** Although promising semantics associates totally ordered timestamps with its messages, interestingly it still allows Coh-CYC's dubious behavior as demonstrated in Figure 4.2a. At execution step $3 thread T_2 writes $X = 1$ at timestamp 2. To certify the promise after $4, the certificate step $\$C_1$ chooses a smaller timestamp (i.e., 1) than that of $3. In contrast, later at step $6 it chooses a larger timestamp for the same write, which eventually leads to the dubious outcome.

Following the same construction as outlined previously, we can simulate the PS execution with the WEAKEST event structure shown in Figure 4.2a. (This is the same event structure as the one shown in Figure 3.3.) We can hence extract the execution shown in Figure 4.2c, which witnesses PS's outcome. □

4.2 Overview of Promising Semantics

As already discussed in §4.1, PS is defined by an abstract machine. Its state $\mathsf{MS} = (\mathcal{TS}, \mathcal{S}, \mathsf{M})$ is a tuple with three components: $\mathcal{TS}$ is the *thread state map*, a function from thread identifiers to thread local states; $\mathcal{S}$ is the *global SC view*, which places a total order on the SC fences; and M is the *shared memory*, represented as a set of messages.

4.2.1 Thread State

The thread state of thread i, $\mathcal{TS}(i)$, is again a tuple $TS = \langle \sigma, V, \mathsf{P} \rangle$ with three components —thread-local state (σ), thread view (V), outstanding promises (P) .

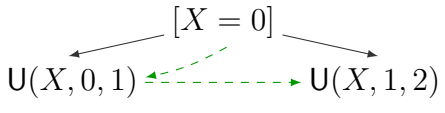
Thread-local state The first component, σ, stores the local state of the thread (e.g., the value of the program counter). As our models are parametric with respect to the syntax of the programming language, we take σ to be a pair of the set of traces of the thread, $\mathbb{P}(i)$, together with the sequence of labels produced so far.

Thread View

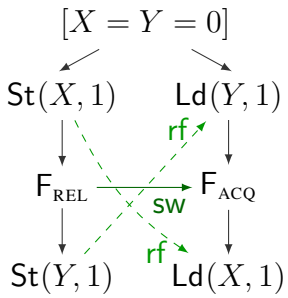
The second component, V, is the *thread view* a mapping from memory locations to the maximum timestamp of a message that the thread is aware of. We already discussed the concept of thread view in §4.1 for programs with relaxed load and store operations. However, in the presence of *release* and *acquire* accesses we refine the concept of thread view. We first consider the release acquire fences and then discuss the release acquire accesses.

Update Access We extend the semantics for the atomic update accesses. In C11 An update atomically performs a read followed by a write on the same location. Consider the following Par-Inc program, taken from Kang et al. [33] with atomic *fetch-and-add*(FAA) update operations. At the end of this program the outcome is $X = 2$ and either $a = 1$ or $b = 1$ based

Initially X=0;
$a = \mathsf{FAA}(X, 1); \parallel b = \mathsf{FAA}(X, 1);$
(Par-Inc)



on the order of the update operations. For instance, in the shown execution above the update in the first thread takes place before the update in the second thread and the execution results in $a = 0 \wedge b = 1$. The alternative execution would result in $a = 1 \wedge b = 0$ and $X = 2$.



(a) C11

Steps	rel	cur	acq	Message
$X = Y = 0$	$X@0, Y@0$	$X@0, Y@0$	$X@0, Y@0$	$X, Y : 0@0$, –
$\mathsf{St}(X, 1)$	”	$X@1, Y@0$	”	$X : 1@1$, $[X@0, Y@0]$
$\mathsf{F}_{\mathrm{REL}}$	$X@1, Y@0$	$X@1, Y@0$	”	–
$\mathsf{St}(Y, 1)$	”	$X@1, Y@1$	”	$Y : 1@1$, $[X@1, Y@0]$
$\mathsf{Ld}(Y, 1)$	$X@0, Y@0$	$X@0, Y@1$	$X@1, Y@1$	–
$\mathsf{F}_{\mathrm{ACQ}}$	”	$X@1, Y@1$	”	–
$\mathsf{Ld}(X, 1)$	”	$X@1, Y@1$	”	–

(b) Promising semantics steps

Figure 4.3: C11 and promising semantics execution for $a = 1 \wedge b = 1$ in MPF program.

To capture the semantics of atomic update, promising semantics refine the concept of timestamp. It considers update as an atomic pair of read followed by write accesses. Hence an update operation has a read timestamp from which immediately precede the write timestamp to on the same location and no write on the same location can have a timestamp between the $(\mathsf{from}, \mathsf{to}]$ range. Note that in the following discussion the to component is the default timestamp of a write operation. Consider the Par-Inc program and the execution where both threads have initial view $[X@0]$ and the memory has initial message $\langle X : 0@0 \rangle$. Now the update in the first thread reserves a timestamp range $(0, 1]$, reads from the message $\langle X : 0@0 \rangle$, and create message $\langle X : 1@1 \rangle$. At the end of the update the view of the first thread is updated to $[X@1]$. Now the update in the second thread takes place which reserves the timestamp $(1, 2]$, reads from the message $\langle X : 1@(0, 1] \rangle$, create message $\langle X : 1@(1, 2] \rangle$, and update the view of the second thread to $[X@2]$.

Views for Fences In this case each thread has three thread views: the *current* view, the *acquire* view, and the *release* view. The current view is the one which we have already discussed earlier. The release and acquire views are used to assign appropriate semantics to release and acquire fences. Thus given a view $V = \langle \mathsf{cur}, \mathsf{rel}, \mathsf{acq} \rangle$, we denote the the current, release, and acquire components by $V.\mathsf{cur}$, $V.\mathsf{rel}$, $V.\mathsf{acq}$ respectively. In a particular thread $V.\mathsf{rel}$ captures the thread's cur view at the point of its last *release* fence. Similarly, $V.\mathsf{acq}$ captures what the thread's cur view will become when it executes an *acquire* fence. Following these rules, $V.\mathsf{rel} \leq V.\mathsf{cur} \leq V.\mathsf{acq}$ holds. In addition, promising semantics attaches the *release view* with a message. Thus, a message now is of the form $m = \langle x : v@t, R \rangle$, where x, v, t are as before, and $m.\mathsf{view} = R$ is the message view where $R(x) \leq t$. Using these views promising semantics captures the effect of synchronization of the C11 model.

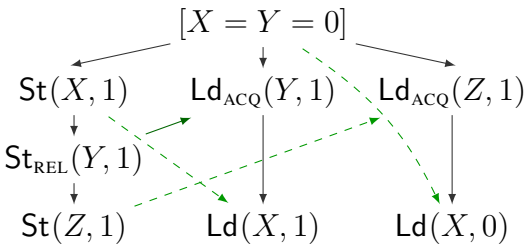
Consider the C11 program:

Initially X=Y=0;

$$\begin{array}{l||l} X = 1; & a = Y; \quad \text{// 1} \\ \mathsf{F}_{\mathrm{REL}}; & \mathsf{F}_{\mathrm{ACQ}}; \\ Y = 1; & b = X; \quad \text{// 1} \end{array} \qquad \text{(MPF)}$$

$X = 1;$
$Y_{\text{REL}} = 1;$
$Z = 1;$
$\Big\|$ $a = Y_{\text{ACQ}};$ // 1
$b = X;$ // $\neq 0$
$\Big\|$ $c = Z_{\text{ACQ}};$ // 1
$d = X;$ // 0

(a) Program



(b) C11 execution

Steps	rel		cur/acq	Message
	Y	Z		
$X = Y = 0$	$X@0, Y@0, Z@0$	$X@0, Y@0, Z@0$	$X@0, Y@0, Z@0$	$X, Y, Z : 0@0, -$
$\mathsf{St}(X, 1)$	"	"	$X@1, Y@0, Z@0$	$X : 1@1$ $-$
$\mathsf{St}_{\text{REL}}(Y, 1)$	$X@1, Y@1, Z@0$	"	$X@1, Y@1, Z@0$	$Y : 1@1$ $[X@1, Y@1, Z@0]$
$\mathsf{St}(Z, 1)$	"	"	$X@1, Y@1, Z@1$	$Z : 1@1$ $[X@0, Y@0, Z@1]$
$\mathsf{Ld}_{\text{ACQ}}(Y, 1)$	$X@0, Y@0, Z@0$	$X@0, Y@0, Z@0$	$X@1, Y@1, Z@0$	$-$
$T_2.\mathsf{Ld}(X, 1)$	"	"	"	$-$
$\mathsf{Ld}_{\text{ACQ}}(Z, 1)$	$X@0, Y@0, Z@0$	$X@0, Y@0, Z@0$	$X@0, Y@0, Z@1$	$-$
$T_3.\mathsf{Ld}(X, 0)$	"	"	"	$-$

(c) Promising semantics steps

Figure 4.4: Example of C11 and promising semantics executions for release acquire accesses.

In this program following the C11 semantics if $a = 1$ then $b = 1$ also holds as shown in the execution in Figure 4.3a. As $\mathsf{Ld}(Y, 1)$ reads from $\mathsf{St}(Y, 1)$, it establishes it establishes synchronization (sw) from F_{REL} to F_{ACQ}. As a result, $\mathsf{St}(X, 1)$ is immediate *happens-before* of $\mathsf{Ld}(X, 1)$ and the load of X in the second thread can only read from the store of the first thread. As a result, when $a = 1$ then $b = 1$ also holds in an execution of this program.

Now we consider the execution in the promising semantics in Figure 4.3b. Consider that the write $X = 1$ in the first thread creates a message $\langle X : 1@1 \rangle$. As a result, when F_{REL} takes place then the release view of the first thread is $[X@1, Y@0]$. Next, write $Y = 1$ results in a message $\langle Y : 1@1, [X@1, Y@0] \rangle$ at time stamp 10 with attached release view $[X@1, Y@0]$. Without the release fence the message would have attached view $[X@0, Y@0]$. In the second thread Y reads from message $\langle Y : 1@1, [X@1, Y@0] \rangle$ and updates the thread's current view to $[X@0, Y@1]$ and acquire view to $[X@1, Y@1]$. Next, the acquire fence updates the current view to $[X@1, Y@1]$ to match with acquire view. At this point if X reads from initialization then the current view of the thread would be $[X@0, Y@1]$ which is not possible as $[X@0, Y@1] < [X@1, Y@1]$. As a result, X reads from message $\langle X : 1@1 \rangle$ and the current view of the second thread remain at $[X@1, Y@1]$.

Views for Release-Acquire Accesses We already know that, in C11, when an acquire read reads-from a release write then it establishes synchronization. Consider the example from

Kang et al. [33] in Figure 4.4 and the respective C11 and promising semantics executions.

In the C11 execution in Figure 4.4b $\mathsf{Ld}_{\text{ACQ}}(Y,1)$ reads from $\mathsf{St}_{\text{REL}}(Y,1)$ and establishes synchronization. Hence $\mathsf{Ld}(X,1)$ can only reads from $\mathsf{St}(X,1)$. On the other hand, there is no synchronization from $\mathsf{St}(Z,1)$ and $\mathsf{Ld}_{\text{ACQ}}(Z,1)$. As a result, $d = X$ can read the initial value of X and create $\mathsf{Ld}(X,0)$.

To model such behaviors, promising semantics refines the release view and captures the release view per location. For instance, in the first thread when $\mathsf{St}_{\text{REL}}(Y,1)$ takes place then the release view on Y is updated to $\mathsf{rel}(Y) = [X@1, Y@1, Z@0]$. In the second thread $\mathsf{Ld}_{\text{ACQ}}(Y,1)$ reads-from $\mathsf{St}_{\text{REL}}(Y,1)$ and updates the current view to $[X@1, Y@1, Z@0]$. As a result $\mathsf{Ld}(X,1)$ is forced to read from $\mathsf{St}(X,1)$. In the third thread $\mathsf{Ld}_{\text{ACQ}}(Z,1)$ reads from the message $Z : 1@1, [X@0, Y@0, Z@1]$ in the first thread with the where the message is created at timestamp $@1$. As a result the current view of the third thread is updated to $[X@0, Y@0, Z@1]$ and in consequence $\mathsf{Ld}(X,0)$ is possible which reads from initial value of X. The execution steps along with views are enlisted in §4.2.1.

Views for Non-atomic Accesses C11 programs also have non-atomic or plain load, store operations on shared memory. Unlike the atomic accesses, non-atomic accesses play no role in synchronization. To address non-atomic accesses, promising semantics refine the thread views and message views further. As a result, given a thread view V, each of the $V.\mathsf{rel}$, $V.\mathsf{cur}$, $V.\mathsf{acq}$ views contain a pair of timestamps $V.\mathsf{rlx}$ and $V.\mathsf{pln}$ for relaxed and non-atomic accesses. Similarly a message view R also contain a pair of $V.\mathsf{rlx}$ and $V.\mathsf{pln}$ timestamps. In a thread with view V, a non-atomic load on location x reads-from a message $\langle x : \_@t \rangle$ when $V.\mathsf{cur}.\mathsf{pln}(x) \leq t$ and then updates the $V.\mathsf{cur}.\mathsf{pln}(x)$ to include t. To add, the load does not include t in the $V.\mathsf{acq}$ view. In case of a non-atomic write, it picks a timestamp t such that $V.\mathsf{cur}.\mathsf{rlx}(x) \leq t$. Moreover, a message created by a non-atomic write carries no view (i.e. $\perp$-view).

Outstanding Promises

The third component, P, records the set of outstanding promises made by thread i; i.e., the set of messages that thread i has added to the shared memory without yet having performed the corresponding writes. In Figure 4.1a we have already shown how promise construct enable $a = b = 1$ outcome in the LB program. The promises have subtle rules to interact with other constructs which we discuss now.

Promise over Same-location Accesses Promising semantics allow to promise over a same-location read operation. However, the read operation is not allowed to read from the promise message as in that case the thread is updated and the promise cannot be fulfilled subsequently. For example, consider the ARM-Weak program from §2.1.3:

$$\begin{array}{c} \text{Initially } X = Y = 0 \\ \begin{array}{l} a = X; \\ X = 1; \end{array} \quad \text{// } 1 \;\Big\|\; X = Y; \;\Big\|\; Y = X; \end{array} \qquad \text{(ARM-Weak)}$$

In this program $a = 1$ is a valid outcome according to the C11 semantics. To get this behavior in the promising semantics, we can promise $X = 1$ in the first thread over the read of X and create a message $\langle X : 1@2 \rangle$. Then the second thread reads-from this message in X and store the value on Y. The third thread reads from Y, store on X at a smaller timestamp than 2, and creates a message say $X : 1@1$. Now $a = X$ reads from this message and update the thread-view to 1 which is smaller than promise timestamp 2. As a result, at each step the promise can have a valid certificate and finally the promise can be fulfilled. As a result, finally the promise execution generates $a = 1$ outcome.

One subtle issue of promise certification is that once a a promise is made the thread must re-recertify at each step that the promise can be fulfilled. For example, in the earlier example, if the write of X in the third thread is performed at a timestamp 3 and creates a message $\langle X : 1@3 \rangle$, then $a = X$ cannot read from the message as in that case the view of the first thread would be updated to $[X@3]$ and the promise can no longer be fulfilled.

Interaction of Promise and C11 Accesses The promising semantics has subtle restrictions on how promise interact with different C11 accesses such as atomic update, release acquire, non-atomic accesses.

We start with the interaction of promising semantics with the atomic updates. The time range reservation for update introduce subtle complexity for the promising semantics as reserving an arbitrary time range for an atomic update may invalidate a promise in some other thread. It is because to get a particular behavior the atomic updates must be performed in a particular order. Hence once we reserve a timerange for a promise, the timerange reservations for the other updates on the same location must be compatible so that the promise can be fulfilled. Hence the semantics introduce a restriction on ensure compatibility of timerange reservation and promise fulfillment; a promise should be fulfillable in *all* future memories. While this rule resolved the interaction between promise and atomic update, we observe that this is too restrictive for certain behaviors which we discuss in the next chapter. More specifically due to this restriction promising semantics to ARMv8 compilation in the presence of atomic updates turns to be unsound. The rule is quite brittle, any change to the rule may affect the results of promising semantics.

Now we discuss the interaction of promises with release/acquire accesses. The promising semantics forbids promises of relaxed writes over fences. It also restricts promises of relaxed writes over same-location release writes. It does, however, allows promises over acquire reads, as well as promises of non-atomic writes over a release fence or a release write on the same location.

4.2.2 Memory

As already mentioned, memory M in promising semantics is just a set of messages, representing the writes (and promises) that have been performed. Each message m records the location written ($m.\mathsf{loc}$), the value written ($m.\mathsf{wval}$), a timestamp ($m.\mathsf{ts}$), as well as some other components. As we discussed earlier the additional components are another timestamp $m.\mathsf{ts}_{\mathsf{f}rom}$, useful for reserving timestamp ranges and giving appropriate semantics to updates, and a mes-

sage view. For a location x, we write $\mathsf{maxmsg}(\mathsf{M}, x)$ for message in M with location x that has the maximum timestamp.

The initial thread state map, $\mathcal{TS}_{\mathsf{init}}(\mathbb{P})$, maps each thread i to the state $\langle \mathbb{P}(i), \epsilon \rangle$, the bottom view (that maps every location to the bottom timestamp), and the empty set of promises. The initial program state, $\mathsf{MS}_{\mathsf{init}}(\mathbb{P})$, has as components the initial thread state map, $\mathcal{TS}_{\mathsf{init}}(\mathbb{P})$, the bottom global SC view, and the initial memory, $\mathsf{M}_{\mathsf{init}}$, that contains an initialization message $\langle x : 0@\ \bot \rangle$ for each location x.

4.2.3 SC-Fence View

PS supports all the operations of our programming language except for SC accesses (reads, writes, and updates). It does support SC fences. Executing an SC fence simply updates the $\mathcal{S}$ component: it computes the elementwise maximal of the global SC view ($\mathcal{S}$) and its thread-local view (V), and updates both components to be that maximal view. In other words, for each location x, it sets both $\mathcal{S}$ and V to the maximum of $\mathcal{S}(x)$ and $V(x)$. Consider the example taken from Kang et al. [33]:

$$\begin{array}{ll} & \text{Initially } X = Y = 0; \\ X = 1; & \| \quad Y = 1; \\ \mathsf{F}_{\mathrm{SC}}; & \| \quad \mathsf{F}_{\mathrm{SC}}; \\ a = Y; \quad // \, 0 & \| \quad b = X; \quad // \neq 0 \end{array}$$

In this program if $a = 0$ then $b = 0$ is not possible due to F_{SC} fences.

Promising semantics ensure this restriction by $\mathcal{S}$ view. Consider that initial views of the two threads are [X@0,Y@0], in the first thread $X = 1$ is performed at timestamp 1 and in the second thread $Y = 1$ is performed at timestamp 1. After these two write the thread views of the first and second threads are $[X@1, Y@0]$ and $[X@0, Y@1]$ respectively. At this point the global view $\mathcal{S}$ is $[X@0, Y@0]$. If the F_{SC} in the first thread is executed then $\mathcal{S}$ view is updated to $[X@1, Y@0]$. Now if the F_{SC} in the second thread updates the $\mathcal{S}$ view as well as the current thread view to $[X@1, Y@1]$. As a result, $a = Y$ may read from initial location and result in $a = 0$, but $b = X$ cannot read 0 from initial location any further as its current thread view is already updated.

4.2.4 More on Promising Semantics

In this section we have informally discussed various examples to demonstrate how the components of a state are affected by read, write, and promise transitions. For complete descriptions of these transitions and the formal definition of the promise machine, we refer the reader to Kang et al. [33].

4.2.5 Program Behavior in Promising Semantics

To define the program behavior in the promising semantics, we first consider the behavior of a promise machine state. We take the behavior of a machine state in a promise execution to

be its final memory contents excluding any unfulfilled promised messages. That is, for each location x, we select the non-promised message in M that has the maximal timestamp and return its value.

$$\mathsf{Behavior}(\mathsf{MS}) \triangleq \{(x, v) \mid x \in \mathsf{Locs} \wedge \mathsf{maxmsg}(\mathsf{MS.M} \setminus \bigcup_i \mathsf{MS}.\mathcal{TS}(i).\mathsf{P}, x).\mathsf{wval} = v\}$$

The behavior of a program $\mathbb{P}$ is the set of behaviors of all *final* machine states that can arise by executing the program (i.e., all machine states reachable from the initial program state that cannot reduce further):

$$\mathsf{Behavior}_{\mathsf{PS}}(\mathbb{P}) \triangleq \{\mathsf{Behavior}(\mathsf{MS}) \mid \mathsf{MS}_{\mathsf{init}}(\mathbb{P}) \rightarrow^* \mathsf{MS} \wedge \mathsf{MS} \not\rightarrow \}\,.$$

We note that recording the final memory contents suffices for distinguishing programs that differ only in their final thread-local states. We can place these programs in a context that writes the contents of the local states to main memory, and then use the definition above to distinguish them.

4.3 Formally Connecting the WEAKEST Model to Promising Semantics

We now state the main result of this section, which says that WEAKEST admits all the program behaviors that PS allows.

Theorem 1. *For a program* $\mathbb{P}$, $\mathsf{Behavior}_{\mathsf{PS}}(\mathbb{P}) \subseteq \mathsf{Behavior}_{\text{WEAKEST}}(\mathbb{P})$.

To prove this theorem, consider a final promise machine state MS of the program $\mathbb{P}$. We have to show that there exists a WEAKEST event structure G of $\mathbb{P}$ with a consistent execution $\mathsf{X} \in \mathsf{ex}_{\text{WEAKEST}}(G)$ such that MS and X have same behavior. Our proof consists of three stages:

1. First, we *define a simulation relation* $G \sim_\Pi \mathsf{MS}$ between the WEAKEST event structure G and the promise machine state MS. The relation is parameterized by a set of threads, Π, in which the event structure is lagging behind MS. The simulation relation relates various components of WEAKEST event structure to certain components of promise machine state by a number of mapping functions. We also define a set of relations to identify the desired execution in the WEAKEST event structure.

2. Next, we prove that steps of the promising machine *preserve* the simulation relation. More precisely, starting from related states, every step of PS can be matched by zero or more steps of the WEAKEST event structure construction yielding related states. To handle promise steps, we split this proof obligation in two lemmas. Lemma 1 considers the transitions by non-promise operations of a particular thread, while Lemma 2 takes care of state transition on entire promise machine state.

3. Finally, the simulation relation is defined so as to also identify an execution X of G such that the behavior of X and MS coincide. So, when PS reaches a final state, we extract that execution from G and complete the proof.

$$\mathsf{spo} \triangleq G.\mathsf{po} \cap (\mathbb{S} \times \mathbb{S}) \qquad \mathsf{srf} \triangleq G.\mathsf{rf} \cap (\mathbb{S} \times \mathbb{S}) \qquad \mathsf{smo} \triangleq \mathsf{mo} \cap (\mathbb{S} \times \mathbb{S})$$
$$\mathsf{sfr} \triangleq (\mathsf{srf}^{-1}; \mathsf{smo}) \setminus [G.\mathsf{E}] \qquad \mathsf{seco} \triangleq (\mathsf{srf} \cup \mathsf{smo} \cup \mathsf{sfr})^+$$
$$\mathsf{ssw} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; \mathsf{spo})^? ; [\mathcal{W}] ; \mathsf{spo}|^?_{\mathsf{loc}} ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; \mathsf{srf}^+ ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (\mathsf{spo} ; [\mathcal{F}])^? ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}]$$
$$\mathsf{shb} \triangleq (\mathsf{spo} \cup \mathsf{ssw})^+$$

Figure 4.5: Auxiliary definitions for the simulation relation.

We move on to the definition of the simulation relation, $G \sim_\Pi \mathsf{MS}$, which says that G simulates MS modulo the threads Π. When $\Pi = \emptyset$, we write $G \sim \mathsf{MS}$ and say that G simulates MS. Our definition uses the following mapping functions, predicates, and relations.

- $\mathbb{W} : (G.\mathsf{E} \cap \mathcal{W}) \rightharpoonup \mathsf{M}$ is a partial function that maps certain write events of G to the corresponding messages in the promise machine memory.
- $\mathbb{S} \subseteq G.\mathsf{E}$ records the set of *covered events* in G, namely the events that are fully executed by the promising machine. In particular, any promised write in $\mathbb{S}$ must have been fulfilled in MS. We write $\mathbb{S}_i$ for the covered events of thread i (i.e., $\mathbb{S} \cap G.\mathsf{E}_i$).
- $\mathsf{sc} \subseteq (\mathbb{S} \cap \mathcal{F}_{\mathrm{SC}}) \times (\mathbb{S} \cap \mathcal{F}_{\mathrm{SC}})$ is a relation that enforces a total order on the SC fences following the $\mathcal{S}$ view of the promise machine. Note that PS disallows SC fence steps to appear in certification runs, which is why $\mathsf{sc} \subseteq \mathbb{S} \times \mathbb{S}$.

Using these relations, we define $\mathsf{mo} \subseteq (G.\mathcal{W} \times G.\mathcal{W})|_{\mathsf{loc}}$ to order the writes to each location x according to the timestamps of the respective messages in the promise machine.

$$\mathsf{mo} \triangleq \{(e_1, e_2) \mid e_1.\mathsf{loc} = e_2.\mathsf{loc} \wedge \mathbb{W}(e_1).\mathsf{ts} < \mathbb{W}(e_2).\mathsf{ts}\} \setminus G.\mathsf{cf}$$

In Figure 4.5, we further define restrictions of the program order, the reads-from relation, the modification order, the reads-before relation, the extended coherence order, the synchronization relation and the happens-before order to events in $\mathbb{S}$. Intuitively, the captured execution $\langle \mathbb{S}, \mathsf{spo}, \mathsf{srf}, \mathsf{smo} \rangle$ corresponds to the promise machine state.

We also define the behavior of the event structure with respect to $\mathbb{W}$ and $\mathbb{S}$ as follows.

$$\mathsf{Behavior}(G, \mathbb{W}, \mathbb{S}) \triangleq \{(x, v) \mid \exists e \in \mathcal{W}_x \cap \mathbb{S}.\ e.\mathsf{wval} = v \wedge \nexists e_1 \in \mathbb{S}.\ \mathsf{mo}(e, e_1)\}$$

Using these auxiliary definitions, we now define the simulation relation as follows.

Definition 6. Let $\mathbb{P}$ be a program with T threads, $\Pi \subseteq \mathsf{T}$ be a subset of threads, G be a WEAKEST event structure, and $\mathsf{MS} = \langle \mathcal{TS}, \mathcal{S}, \mathsf{M} \rangle$ be a promise machine state. We say that $G \sim_\Pi \mathsf{MS}$ holds iff there exist $\mathbb{W}$, $\mathbb{S}$, and sc such that the following conditions hold:

1. G is consistent according to the WEAKEST model: $\mathsf{isCons}_{\mathrm{WEAKEST}}(G)$.
2. The local state of each thread in MS contains the program of the thread along with the sequence of covered events of that thread: $\forall i.\ \mathcal{TS}(i).\sigma = \langle \mathbb{P}(i), \mathit{labels}(\mathsf{sequence}_{\mathsf{spo}}(\mathbb{S}_i)) \rangle$.

3. Whenever $\mathbb{W}$ maps an event of G to a message in MS, then the location accessed and the written values match: $\forall e \in \mathsf{dom}(\mathbb{W}).\ e.\mathsf{loc} = \mathbb{W}(e).\mathsf{loc} \wedge e.\mathsf{wval} = \mathbb{W}(e).\mathsf{wval}$.

4. All outstanding promises of threads $(\mathsf{T} \setminus \Pi)$ have corresponding write events in G that are po-after $\mathbb{S}$: $\forall i \in (\mathsf{T} \setminus \Pi).\ \forall e \in (\mathbb{S}_0 \cup \mathbb{S}_i).\ \mathcal{TS}(i).\mathsf{P} \subseteq \{\mathbb{W}(e') \mid (e, e') \in G.\mathsf{po}\}$.

5. For every location x and thread i, the thread view of x in the promise state MS records the timestamp of the maximal write visible to the covered events of thread i.

$$\forall i, x.\ \mathcal{TS}(i).V(x) = \mathsf{max}\{\mathbb{W}(e).\mathsf{ts} \mid e \in \mathsf{dom}([\mathcal{W}_x]; G.\mathsf{jf}^?; \mathsf{shb}^?; \mathsf{sc}^?; \mathsf{shb}^?; [\mathbb{S}_i])\}$$

6. The $\mathbb{S}$ events satisfy coherence: $\mathsf{shb}; \mathsf{seco}^?$ is irreflexive.

7. The atomicity condition holds for the $\mathbb{S}$ events: $\mathsf{sfr}; \mathsf{smo}$ is irreflexive.

8. The SC fences are appropriately ordered by sc: $[\mathcal{F}_{\mathrm{SC}}]; (\mathsf{shb} \cup \mathsf{shb}; \mathsf{seco}; \mathsf{shb}); [\mathcal{F}_{\mathrm{SC}}] \subseteq \mathsf{sc}$.

9. The behavior of MS matches that of the $\mathbb{S}$ events: $\mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(G, \mathbb{W}, \mathbb{S})$.

Next, we establish Lemmas 1 and 2, which concern the preservation of the simulation relation by program steps. First, we show that non-promising steps of threads i preserve simulation modulo thread i.

Lemma 1. $G \sim_{\{i\}} \mathsf{MS} \wedge \mathsf{MS} \xrightarrow{np}_i \mathsf{MS}' \implies \exists G'.\ G \to_{\mathbb{P},\mathrm{WEAKEST}}{}^* G' \wedge G' \sim_{\{i\}} \mathsf{MS}'$.

We prove this lemma in Chakraborty [21, Appendix A] by case analysis over the promise machine state transition. We simply perform the corresponding event structure construction step and define updated $\mathbb{W}$, $\mathbb{S}$, and sc to show that $G' \sim_{\{i\}} \mathsf{MS}'$ holds.

Using this lemma, we establish the following stronger property:

Lemma 2. $G \sim \mathsf{MS} \wedge \mathsf{MS} \to \mathsf{MS}' \implies \exists G'.\ G \to_{\mathbb{P},\mathrm{WEAKEST}}{}^* G' \wedge G' \sim \mathsf{MS}'$.

To prove Lemma 2, we case split on the operation of the transition $\mathsf{MS} \to \mathsf{MS}'$. In case of a non-promise operation we construct G' in two steps. First, we first use Lemma 1 to establish $G_1 \sim_{\{i\}} \mathsf{MS}'$ for an appropriate G_1. Then, in the next step, we consider the PS certification run of thread i, $\mathsf{MS}' \xrightarrow{np}_i{}^* \mathsf{MS}''$ with $\mathsf{MS}''.\mathcal{TS}(i).\mathsf{P} = \emptyset$. We inductively apply Lemma 1 on this certification run to extend G_1 to G' such that $G' \sim_{\{i\}} \mathsf{MS}''$. Since, however, MS'' has no outstanding promises for thread i, it follows that $G' \sim \mathsf{MS}''$, and consequently also $G' \sim \mathsf{MS}'$ (for a smaller $\mathbb{S}$), as required. In case of a promise operation, we just consider the PS certification run of thread i and consequently establish $G' \sim \mathsf{MS}'$. The proof is discussed in Chakraborty [21, Appendix A].

Finally, Theorem 1 follows from Lemma 2 by induction as proved below.

Proof. To state Theorem 1 formally,

$$\forall \mathbb{P}.\ \forall \mathsf{MS}.\ (\mathsf{MS}_{\mathsf{init}}(\mathbb{P}) \to^* \mathsf{MS} \wedge \mathsf{MS} \not\to).\ \exists G, \mathsf{X}.\ G_{\mathsf{init}} \to_{\mathbb{P},\mathrm{WEAKEST}}{}^* G \wedge \mathsf{X} \in \mathsf{ex}_{\mathrm{WEAKEST}}(G).\ \wedge \mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(\mathsf{X})$$

Step 1. Given a program $\mathbb{P}$, from Lemma 2 we show that using the simulation relation in Definition 6, we can follow the promise machine steps and for a promise machine state state MS we can construct an WEAKEST event structure G, that is, $G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKEST}}{}^{*} G$.

Step 2. Now we extract a consistent execution X from G where $\mathsf{X} \in \mathsf{ex}_{\text{WEAKEST}}(G)$, such that $\mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(\mathsf{X})$.

Given the event structure G along with $\mathbb{S}$ and related sets, the execution $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ is as follows: $\mathsf{X.E} = \mathbb{S}$, $\mathsf{X.po} = \mathsf{spo}$, $\mathsf{X.rf} = \mathsf{srf}$, and $\mathsf{X.mo} = \mathsf{smo}$. Note that the events in X.E is conflict-free as $\mathbb{S}$ is conflict-free in G. Now we check whether execution X is consistent.

- from the definitions of spo, srf, smo, we know $\mathsf{X.po} \subseteq (\mathbb{S} \times \mathbb{S})$, $\mathsf{X.rf} \subseteq (\mathbb{S} \times \mathbb{S})$, and $\mathsf{X.mo} \subseteq (\mathbb{S} \times \mathbb{S})$. Hence X is (Well-formed).
- From the definition, we know smo is total as the order on the timestamps on the same location is total in the promise machine. Hence X.mo is total and (total-MO) holds in execution X.
- From the construction of G we know that $\mathsf{shb}; \mathsf{seco}^{?}$ is irreflexive. Hence $(\mathsf{X.hb}_{\mathsf{C11}}; \mathsf{X.eco}^{?})$ is irreflexive and (Coherence) holds in G.
- From the construction we know that $[G.\mathsf{U} \cap \mathbb{S}]; (\mathsf{sfr}; \mathsf{smo}) = \emptyset$ holds. From the definition we know that $\mathsf{X.U} = (G.\mathsf{U} \cap \mathbb{S})$, $\mathsf{X.fr} = \mathsf{sfr}$, and also $\mathsf{X.mo} = \mathsf{smo}$ holds. Hence $[\mathsf{X.U}]; (\mathsf{X.fr}; \mathsf{X.mo}) = \emptyset$ hold and X preserves (Atomicity).
- From the simulation relation in the construction we know that sc is total in G and $[G.\mathcal{F}_{\text{SC}}]; \mathsf{shb} \cup \mathsf{shb}; \mathsf{seco}; \mathsf{shb}; [G.\mathcal{F}_{\text{SC}}] \subseteq \mathsf{sc}$ holds. Hence irreflexivity holds for the relation $[G.\mathcal{F}_{\text{SC}}]; \mathsf{shb} \cup \mathsf{shb}; \mathsf{seco}; \mathsf{shb}; [G.\mathcal{F}_{\text{SC}}]$. From definition we know that $\mathsf{X}.\mathcal{F}_{\text{SC}} = G.\mathcal{F}_{\text{SC}}$, $\mathsf{X.hb}_{\mathsf{C11}} = \mathsf{shb}$, and $\mathsf{X.eco} = \mathsf{seco}$ hold. As a result, irreflexivity holds for $\mathsf{X.psc}_{\mathsf{F}} = [\mathsf{X}.\mathcal{F}_{\text{SC}}]; \mathsf{X.hb}_{\mathsf{C11}} \cup \mathsf{X.hb}_{\mathsf{C11}}; \mathsf{X.eco}; \mathsf{X.hb}_{\mathsf{C11}}; [\mathsf{X}.\mathcal{F}_{\text{SC}}]$. Note that X does not have any SC memory access and hence $\mathsf{X.psc}_{\mathsf{base}} = \emptyset$. Hence X preserves (SC).

Thus X is consistent and hence $\mathsf{X} \in \mathsf{ex}_{\text{WEAKEST}}(G)$.

Step 3. From the construction we know that $\mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(G, \mathbb{W}, \mathbb{S})$. Hence from the definitions $\mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(\mathsf{X})$.

Thus considering step 1, 2, 3 the theorem holds. □

Chapter Summary In this chapter we have shown that WEAKEST is strictly weaker than PS. We have formally proved the connection between WEAKEST and PS.

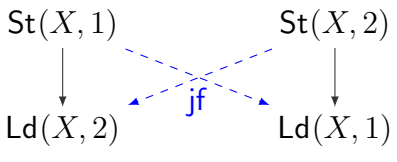
However, we also observe that both PS and WEAKEST model allow some out-of-thin-air execution as shown in Figure 4.2 for the Coh-CYC program. To avoid such invalid outcome we propose a stronger model called WEAKESTMO in the next chapter.

5 The WEAKESTMO Memory Model

In this chapter, we define the WEAKESTMO model. The WEAKESTMO model strengthens the WEAKEST model by introducing modification-order (mo) between same-location writes as a primitive in the event structure. The purpose of recording mo is to rule out coherence violations at the event structure level, thereby yielding a stronger model than WEAKEST (i.e., it allows fewer outcomes).

Recall the Coh program and Coh-ES WEAKEST event structure from §3.2.2.

$$X = 1; \quad a = X; \ // 2 \;\Big\|\; X = 2; \quad b = X; \ // 1$$



The WEAKESTMO model requires all non-conflicting stores to the same location to be related by mo. Suppose, without loss of generality, that we have an mo-edge from $\mathsf{St}(X,1)$ to $\mathsf{St}(X,2)$. Then, the load $\mathsf{Ld}(X,1)$ is *incoherent* because it reads an overwritten value, and so WEAKESTMO forbids that event structure. By symmetry, WEAKESTMO also rules out the event structure, where mo goes in the other direction.

5.1 Formalization: The WEAKESTMO Model

We move on to the formalization of the WEAKESTMO model. We discuss the WEAKESTMO event structure construction steps, consistency constraints, and related details which extend those of WEAKEST model.

5.1.1 WEAKESTMO Event Structures

A WEAKESTMO event structure is of the form $G \triangleq \langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo} \rangle$ where the E, po, jf and ew components are exactly the same as those of WEAKEST event structures. The last component, $\mathsf{mo} \subseteq ((\mathsf{E} \cap \mathcal{W}) \times (\mathsf{E} \cap \mathcal{W}))|_{\mathsf{loc}}$, records the *modification order*, a strict partial order that orders write operations on the same memory location. We require that mo is total on non-conflicting writes to the same location and that equal writes have the same mo-successors, i.e., $G.\mathsf{ew} \,;\, G.\mathsf{mo} \subseteq \mathsf{mo}$.

Auxiliary Definitions We derive the following relations in the WEAKESTMO model which are same as those of the WEAKEST model as follows.

$$G.\mathsf{sw} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; G.\mathsf{po})^? ; [\mathcal{W}] ; G.\mathsf{po}|_{\mathsf{loc}}^? ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; G.\mathsf{jf}^+ ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (G.\mathsf{po} ; [\mathcal{F}])^? ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}] \quad \text{(Synchronizes-with)}$$

$$G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup G.\mathsf{sw})^+ \quad \text{(Happens-before)}$$

$$G.\sim \triangleq G.\mathsf{cf} \setminus (G.\mathsf{cf} ; G.\mathsf{po} \cup G.\mathsf{po}^{-1} ; G.\mathsf{cf}) \quad \text{(Immediate conflict)}$$

$$G.\mathsf{ecf} \triangleq (G.\mathsf{hb}^{-1})^? ; G.\mathsf{cf} ; G.\mathsf{hb}^? \quad \text{(Extended conflict)}$$

$$G.\mathsf{rf} \triangleq (G.\mathsf{ew}^? ; G.\mathsf{jf}) \setminus G.\mathsf{cf} \quad \text{(Reads-from relation)}$$

In order to reason about coherence, we also define two relations—*reads-before/from-read* (fr) and the *extended coherence order* (eco)—exactly as in RC11 [37]. These relations are similar to the $G.\mathsf{fr}_{\mathsf{strong}}$, $G.\mathsf{eco}_{\mathsf{strong}}$ relations in the WEAKEST model where $G.\mathsf{mo}_{\mathsf{strong}}$ corresponds to the $G.\mathsf{mo}$ in the WEAKESTMO event structure. A read event r reads before a write event w if r reads from some write w' that is mo-after w. (In the formal definition, we subtract the identity relation so as to avoid saying that an update reads before itself.) Finally, the extended coherence order is the transitive closure of the union of the three coherence-enforcing relations (rf, mo, and fr).

$$G.\mathsf{fr} \triangleq (G.\mathsf{rf}^{-1} ; G.\mathsf{mo}) \setminus [\mathcal{E}] \quad \text{(Reads-before/from-read relation)}$$

$$G.\mathsf{eco} \triangleq (G.\mathsf{rf} \cup G.\mathsf{mo} \cup G.\mathsf{fr})^+ \quad \text{(Extended coherence order)}$$

5.1.2 WEAKESTMO Consistency Constraints

Now we move to the consistency constraints in the WEAKESTMO model. The consistency constraints in WEAKESTMO are same as the WEAKEST model except the coherence constraint where we replace (COH) by (COH′).

Definition 7. An event structure G is *consistent* according to WEAKESTMO model, written $\mathsf{isCons}_{\mathrm{WEAKESTMO}}(G)$, iff the following conditions hold:

(CF) No event can be in extended conflict with itself: $G.\mathsf{ecf}$ is irreflexive.

(CFJ) A write cannot justify a read in extended conflict: $G.\mathsf{jf} \cap G.\mathsf{ecf} = \emptyset$.

(VISJ) Only visible events justify reads of other threads: $\mathsf{dom}(G.\mathsf{jfe}) \subseteq \mathsf{vis}(G)$.

(ICF) Immediately conflicting events must be reads: $G.\sim \subseteq \mathcal{R} \times \mathcal{R}$.

(ICFJ) Immediately conflicting reads cannot be justified by the same or by equal writes: $G.\mathsf{jf} ; G.\sim ; G.\mathsf{jf}^{-1} ; G.\mathsf{ew}^?$ is irreflexive.

(COH′) $G.\mathsf{hb} ; G.\mathsf{eco}^?$ is irreflexive.

(NCFU) $[G.\mathsf{U}_{\sqsupseteq \mathrm{ACQ}}] ; G.\mathsf{jf}^{-1} ; G.\mathsf{rf} ; [G.\mathsf{U}_{\sqsupseteq \mathrm{ACQ}}] \subseteq \mathsf{cf}^?$.

$$A \subseteq G.\mathsf{E}_{e.\mathsf{tid}} \qquad \mathsf{dom}([\mathsf{E}_{e.\mathsf{tid}}]\,;\mathsf{po}\,;[A]) \subseteq A \qquad \mathsf{labels}(\mathsf{sequence}_{\mathsf{po}}(A)) \cdot e.\mathsf{lab} \in \mathbb{P}(e.\mathsf{tid})$$
$$\mathsf{E}' = \mathsf{E} \uplus \{e\} \quad \mathsf{po}' = \mathsf{po} \cup (A \times \{e\}) \quad \mathsf{isCons}_M(\langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle) \quad CF = (\mathsf{E}_{e.\mathsf{tid}} \setminus A)$$
$$\textbf{if } e \in \mathcal{R} \textbf{ then } \exists w \in \mathsf{E} \cap \mathcal{W}.\ \mathsf{jf}' = \mathsf{jf} \cup \{(w,e)\} \wedge w.\mathsf{loc}{=}e.\mathsf{loc} \wedge w.\mathsf{wval}{=}e.\mathsf{rval} \textbf{ else } \mathsf{jf}'{=}\mathsf{jf}$$
$$\textbf{if } e \in \mathcal{W}_{\sqsubseteq \mathrm{RLX}} \textbf{ then } \mathsf{AddEW}(\mathsf{ew}, \mathsf{ew}', CF, e) \textbf{ else } \mathsf{ew}' = \mathsf{ew}$$
$$\textbf{if } e \in \mathcal{W} \textbf{ then } \mathsf{AddMO}(\mathsf{mo}, \mathsf{mo}', \mathsf{E}, CF, \mathsf{ew}, e) \textbf{ else } \mathsf{mo}' = \mathsf{mo}$$
$$\langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo} \rangle \rightarrow_{\mathbb{P},M} \langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle$$

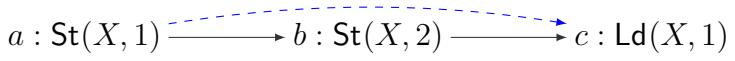
Figure 5.1: One construction step of a program's WEAKESTMO event structure which extends WEAKEST construction step in Figure 3.6 with AddMO.

(NCFSC) $G.\mathsf{pscb} \cup G.\mathsf{pscf}$ is acyclic.

where (as in the WEAKEST model) $G.\mathsf{jfe} \triangleq G.\mathsf{jf} \setminus G.\mathsf{po}$ and

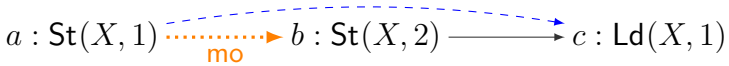
$$\mathsf{vis}(G) \triangleq \{e \in G.\mathsf{E} \mid [\mathcal{W}];(G.\mathsf{cf} \cap G.\mathsf{jfe};(G.\mathsf{po} \cup G.\mathsf{jf})^*;G.\mathsf{jfe};G.\mathsf{po}^?);[\{e\}] \subseteq G.\mathsf{ew};G.\mathsf{po}^=\}\,.$$

We have already discussed the (CF), (CFJ), (VISJ), (ICF), (ICFJ) consistency constraints in §3.3.1. WEAKESTMO introduces the (COH′) constraint by replacing the (COH) constraint of the WEAKEST model. These two coherence constraints (COH) and (COH′) differ only in the presence of concurrent writes to the same location; i.e., of two writes to a given location that are not ordered by hb. Recall the example we discussed earlier in the conext of WEAKEST model:



(Basic coherence violation)

If, for example, in the event structure above, the events a and b were not related by po, WEAKEST would allow the justification edge. WEAKESTMO would also allow it but only if the mo-edge went from b to a. If mo went from a to b, as depicted below, the event structure would be inconsistent:



(Coherence violation)

Thus the (COH′) enforces a stronger coherence constraint than (COH). As already explained in §5.2, the difference is evident in programs Coh and Coh-CYC.

Finally, we define two additional constraints (NCFU) and (NCFSC) for WEAKESTMO event structure. The (NCFU) constraint provides an weaker atomicity constraints; that is, a pair of non-conflicting acquire or stronger atomic updates cannot read from same write operation. The (NCFSC) constraint orders the SC accesses at the event structure level. Essentially these two constraints ensure that a conflict-free event structure is an RC11 consistent execution when the execution is *RLX-race-free* (cf. §6.2).

5.1.3 WEAKESTMO Event Structure Construction

The event structure construction in WEAKESTMO is shown in Figure 5.1. Note that the construction is similar to that of WEAKEST model with subtle extension for mo relations. The

construction updates the mo relations by AddMO as follows:

$$\begin{aligned}\mathsf{AddMO}(\mathsf{mo},\mathsf{mo}',\mathsf{E},CF,\mathsf{ew},e) \triangleq \exists w \in \mathcal{W} \cap \mathsf{E} \setminus CF.\ w.\mathsf{loc} = e.\mathsf{loc} \wedge \\ \mathsf{mo}' = \mathsf{mo} \cup (\mathsf{dom}(\mathsf{mo}^?;\mathsf{ew}^?;[\{w\}]) \times \{e\}) \\ \cup (\{e\} \times \mathsf{codom}([\{w\}];\mathsf{mo};\mathsf{ew}^?))\end{aligned}$$

AddMO selects a write event w in the graph that writes to the same location as e and does not conflict with it. Since the event structure is initialized, such a write always exists. It then puts e immediately after w in mo.

5.1.4 Execution Extraction in the WEAKESTMO Model

The definition of consistent execution in WEAKESTMO is same as that of WEAKEST model as described in §3.3.3. Similar to WEAKEST, we extract a set of execution after the construction of a consistent event structure in WEAKESTMO. Formally,

$$\begin{aligned}\mathsf{GoodRestriction}(G) &\triangleq \left\{A \;\middle|\; A \subseteq \mathsf{vis}(G) \wedge [A]\,;G.\mathsf{cf}\,;[A] = \emptyset \wedge \mathsf{dom}(G.\mathsf{hb}\,;[A]) \subseteq A\right\} \\ \mathsf{Project}_{\text{WEAKESTMO}}(G,A) &\triangleq \{\langle A\,, G.\mathsf{po} \cap (A\times A)\,, G.\mathsf{rf} \cap (A\times A)\,, G.\mathsf{mo} \cap (A\times A)\rangle\}\end{aligned}$$

The definition of $\mathsf{GoodRestriction}(G)$ is same as in WEAKEST model. Projection to an execution is trivial under WEAKESTMO: we just restrict po, rf, and mo to the set of events, A. Similar to WEAKEST, we extract a set of consistent execution from a WEAKESTMO consistent event structure.

$$\mathsf{ex}_{\text{WEAKESTMO}}(G) \triangleq \left\{\mathsf{X} \;\middle|\; \begin{array}{c}\exists A \in \mathsf{GoodRestriction}(G).\ \mathsf{X} \in \mathsf{Project}_{\text{WEAKESTMO}}(G,A) \\ \wedge\ \mathsf{isCons}_{\text{WEAKESTMO}}(\mathsf{X})\end{array}\right\}$$

5.1.5 Program Behaviors

The execution behavior in WEAKESTMO is same as that of WEAKEST model, that is, i.e., the value written by the mo-maximal write for each location.

$$\mathsf{Behavior}(\mathsf{X}) \triangleq \{(e.\mathsf{loc}, e.\mathsf{wval}) \mid \exists e \in \mathsf{X.E} \cap \mathcal{W}.\ [\{e\}]\,;\mathsf{X.mo} = \emptyset\}$$

Then, we adapt the definition of program behaviors under WEAKEST to the WEAKESTMO model as follows.

$$\begin{aligned}\mathsf{Behavior}_{\text{WEAKESTMO}}(\mathbb{P}) \triangleq \{\mathsf{Behavior}(\mathsf{X}) \mid \exists G.\ \ G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKESTMO}}{}^{*}\ G\ \wedge\ \mathsf{X} \in \mathsf{ex}_{\text{WEAKESTMO}}(G) \\ \wedge\ \mathsf{maximal}_{\mathbb{P}}(\mathsf{X})\}\end{aligned}$$

where $\mathsf{maximal}_{\mathbb{P}}(\mathsf{X}) \triangleq \nexists i, lab.\ \mathsf{labels}(\mathsf{sequence}_{\mathsf{X.po}}(\mathsf{X.E}_i))\cdot lab \in \mathbb{P}(i)$.

Similar to WEAKEST, $\mathsf{Behavior}_{\text{WEAKESTMO}}(\mathbb{P})$ contains the behaviors of any *maximal* execution X extracted from an event structure G that was constructed from the program $\mathbb{P}$ where *maximality* ensures that all threads have terminated according to the thread semantics.

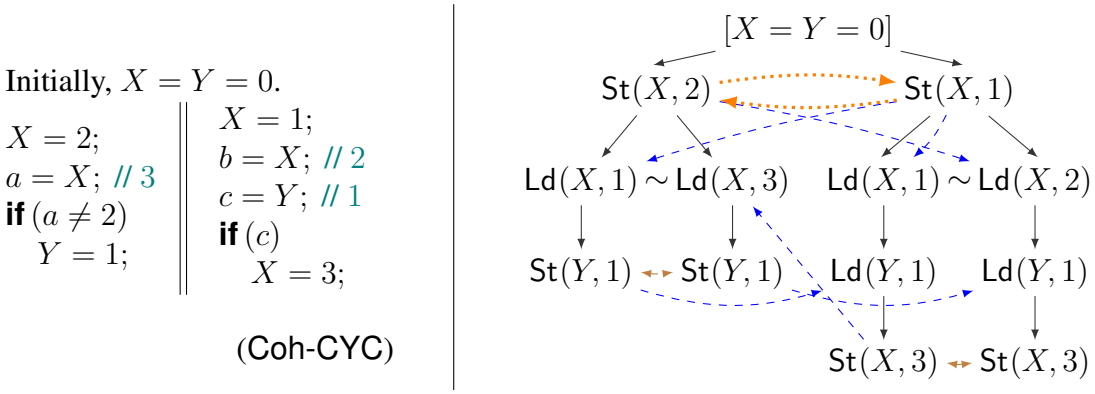


Figure 5.2: WEAKESTMO discards the WEAKEST event structure of Coh-CYC program.

5.2 WEAKESTMO and Promising Semantics

We observe that the WEAKESTMO and the promising semantics (PS) [33] is incomparable. There are certain program behaviors which are allowed in PS but disallowed in WEAKESTMO and there are certain program behaviors which are allowed in WEAKESTMO but PS discards.

Program Behavior allowed in PS but not in WEAKESTMO Consider the Coh-CYC program and the WEAKEST event structure in Figure 3.3 which yields an execution with outcome $a = 3 \wedge b = 2 \wedge c = 1$. As shown in Figure 5.2, the Ld$(X, 1)$, Ld$(X, 2)$ reads imply mo cycle between St$(X, 1)$ and St$(X, 2)$ and in turn contains the incoherent Coh-ES event structure from § 3.2.2 as a substructure. Hence WEAKESTMO discards this WEAKEST event structure. As a result, WEAKESTMO disallows the $a = 3 \wedge b = 2 \wedge c = 1$ outcome in Coh-CYC program which is allowed in PS (see Figure 4.2a).

Program Behavior allowed in WEAKESTMO but not in PS We move on to an example with atomic updates: the program FADD shown in Figure 5.3. This is another variant of the load-buffering program, where the store to Y on thread T_1 depends on the result of a fetch-and-add instruction. The final value of Z therefore depends on the previous load of X, but the return value of fadd does not. As a result, the ARMv8 model [61] allows the annotated weak outcome $a = c = 1$.

Both our models, WEAKEST and WEAKESTMO allow the same outcome with the event structure displayed in Figure 5.3. The execution obtained by extracting the events of the second branch of T_1 and the events of T_2 is consistent and witnesses the discussed outcome.

Nevertheless, as we will shortly see, PS forbids this outcome. In order to handle atomic updates, PS has a few more features that we have not yet mentioned. Specifically, instead of a single timestamp, PS messages carry a timestamp range $(\mathsf{from}, \mathsf{to}]$. Generally, different messages in memory have disjoint timestamp ranges; so whenever two messages to Z have adjacent messages, this means that there cannot be another message to X in between, neither in the current memory nor in any future extension of that memory. This feature is used to satisfy the key invariant of atomic updates, namely that they read from their immediately preceding

Initially, $X = Y = Z = 0$.

$a = X$; // reads 1	$c = Y$; // reads 1
$b = \mathsf{fadd}(Z, a)$;	$X = c$;
$Y = b + 1$;	

(FADD)

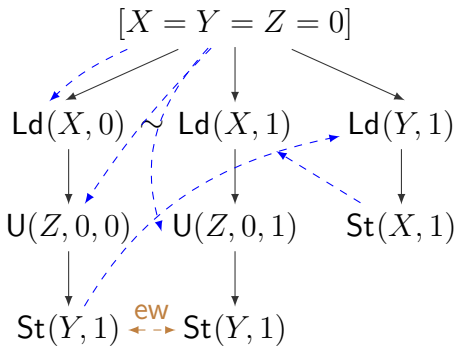


Figure 5.3: The FADD program whose $a = c = 1$ outcome is allowed by ARMv8, WEAKEST, and WEAKESTMO. The instruction $\mathsf{fadd}(Z, a)$ atomically increments the value at location Z by a, and returns the old value of Z.

write in modification order. Further, to ensure that an execution never gets stuck because of an update whose slot is taken by another thread, PS has a more sophisticated certification condition. It requires that the outstanding promises of each thread be thread-locally certifiable not only in the present memory but also in *every future memory* (i.e., in every memory that extends the current one with possibly more messages).

Now consider running FADD under PS. To get the outcome $a = c = 1$, we must start with a promise. Clearly, the machine cannot promise $X = 1$ nor $Z = 1$ because these writes cannot be certified in the current memory. However, it also cannot promise $Y = 1$ in T_1 because there exists a future memory, namely one with the message $\langle Z : 42@(0, 1]\rangle$, where T_1 cannot fulfill its promise (it will write $Y = 43$). As a result, PS cannot produce the outcome $a = c = 1$, which in turn means that its intended compilation to ARMv8 is unsound. To restore soundness of compilation from PS to ARMv8, Podkopaev et al. [59] insert a `dmb.ld` fence after every atomic update.

In summary, we have shown that WEAKEST is strictly weaker than both WEAKESTMO and PS, while WEAKESTMO and PS are incomparable. Although our models further correct the aforementioned counterexample of PS's compilation to ARMv8, we so far do not have proof of soundness of compilation from WEAKESTMO to ARMv8.

5.3 LLVM Concurrency Formalization

Now we use WEAKESTMO model to formalize the relaxed memory concurrency semantics in LLVM, specifically for racy programs. Before going into the details we discuss data races.

5.3.1 Data Race

In the context of an execution X, we say that two non-conflicting events are *concurrent* if they are not related by the happens-before relation. Two concurrent events are *racy* if they access the same location and at least one of them is a write. Let X.Race be the set of all racy events

of execution X:

$$\mathsf{X.Race} \triangleq \mathsf{dom}(((\mathsf{X.E} \times \mathsf{X.E}) \setminus (\mathsf{X.hb}^{=}_{\mathsf{C11}} \cup \mathsf{X.cf}))|_{\mathsf{loc}} \cap \mathsf{one}(\mathcal{W}))$$

where $\mathsf{one}(A)$ is the relation saying that at least one of its components belongs to the set A; that is, $\mathsf{one}(A)(x, y) \triangleq (x \in A \vee y \in A)$.

In this context we consider the data races where atleast one of the accesses is non-atomics. that is, $\mathsf{X.Race} \subseteq \mathsf{Ld}_{\mathrm{NA}} \cup \mathsf{St}_{\mathrm{NA}}$. According to the C11 semantics [30, 29], the behavior of a program which has a consistent execution with race on non-atomic access is undefined. However, the semantics of LLVM concurrency differs from C11 concurrency semantics.

5.3.2 Relaxed Memory Concurrency Semantics in LLVM

We already know that LLVM introduces similar set of concurrency primitives with certain exceptions. LLVM 'monotonic' memory order which is same as C11 'relaxed' memory order. Moreover, LLVM introduces 'unordered' memory order which is unused in C11 compilation. Hence we ignore 'unordered' access in this thesis.

The LLVM concurrency semantics differs from C11 in handling racy programs [22]. However, before discussing the differences with discuss 'undef' expression in LLVM.

The Semantics of the Undefined Value

In LLVM, the special undefined value $\mathbf{u}$ is introduced as the result of erroneous computations, such as reading from an uninitialized memory location as a replacement of an arbitrary constant value. This special value propagates through every assignment and arithmetic operation. So, for example, $\mathbf{u} + 1 \rightsquigarrow \mathbf{u}$ and even $\mathbf{u} * 0 \rightsquigarrow \mathbf{u}$.[1]

The intended semantics is that the compiler may replace $\mathbf{u}$ with any concrete value it finds most convenient, and that moreover different uses of the same $\mathbf{u}$ may be even replaced by different values by the compiler. This weak semantics leads to some rather unexpected behaviors. For example,

$$\begin{array}{l} \mathsf{int}\ t; \\ \mathsf{if}(t \leq 1\ \&\&\ t > 1) \\ \quad \mathsf{printf}(\text{“Hi”}); \end{array} \qquad (\mathsf{UCond})$$

may print "Hi" even though the if-condition seems unsatisfiable. The reason is that t is uninitialized and hence returns $\mathbf{u}$ in each use, which can be used to satisfy the condition.

Strange though it may seem, LLVM's treatment of uninitialized reads is allowed by the C standard, which says that performing any computation with a value returned by an uninitialized read results in *undefined* behavior.

[1] This is needed to justify the distributivity of $+$ over $*$. Consider the transformations: $\mathbf{u} * 0 \rightsquigarrow \mathbf{u} * (1 - 1) \rightsquigarrow \mathbf{u} * 1 + \mathbf{u} * (-1) \rightsquigarrow \mathbf{u} + \mathbf{u} \rightsquigarrow \mathbf{u}$.

The Semantics of Data Races in LLVM

LLVM distinguishes between *write-write* races and *read-write* races. A write-write race is one happening between a pair of write events, whereas a read-write race is between a load event and a concurrent write event.

According to LLVM, only write-write NA-races result in undefined behavior. In contrast, read-write NA-races have defined behavior: the racy read may return an $\mathbf{undef}(\mathbf{u})$ expression which can be materialized to an arbitrary constant value of the same data type. Consider the following program where initially $Y = 0$.

$$Y_{\text{NA}} = 1; \;\Big\|\; \begin{array}{l} t = Y_{\text{NA}}; \\ \mathsf{if}(t \leq 1 \,\&\&\, t > 1) \\ \quad \mathsf{printf}(\text{“Hi”}); \end{array}$$

As with the UCond program in §5.3.2, the current program may also print “Hi” just because the non-atomic load of Y is racy and thus returns $\mathbf{u}$. The reason that the treatment of read-write races in the LLVM semantics differs from that in C11 is because LLVM readily performs the following transformation

$$\begin{array}{l} \mathsf{if}(cond) \\ \quad t = X_{\text{NA}}; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t' = X_{\text{NA}}; \\ t = cond \,?\, t' : t; \end{array}$$

that converts a conditional branch into a conditional move instruction. This transformation may, however, introduce a read-write race if there were some other parallel thread writing to X only when the condition $cond$ is false. The transformation is correct because the target execution uses the racy read value only when the source execution is also racy.

A write-write race occurs whenever both of the accesses racing with one another are stores or updates. In this case, the intended semantics according to the LLVM documentation [43, Section “Optimization outside atomic”] is the same as in C11: even a single consistent execution with a write-write race results in the program having *undefined behavior*. This semantics allows the read-after-write elimination over an acquire access as shown in the following example:

$$\begin{array}{l} X_{\text{NA}} = 4; \\ \mathsf{if}(Y_{\text{ACQ}}) \\ \quad t = X_{\text{NA}}; \end{array} \;\Big\|\; \begin{array}{l} X_{\text{NA}} = 8; \\ Y_{\text{REL}} = 1; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X_{\text{NA}} = 4; \\ \mathsf{if}(Y_{\text{ACQ}}) \\ \quad t = 4; \end{array} \;\Big\|\; \begin{array}{l} X_{\text{NA}} = 8; \\ Y_{\text{REL}} = 1; \end{array}$$

Because of the write-write race on X, the source program has undefined behavior, and hence the transformation is trivially sound. If, however, write-write races were not considered to be undefined behavior, but rather that one of the accesses occurred before the other, then the transformation would be unsound, because in the source program, t would have to contain the final value of X (which may well be 8).

This optimization was performed by LLVM version 3.6 but was later dropped in version 3.7 while fixing another concurrency bug (LLVM Bug #22514 [42]). This demonstrates that it is important for LLVM to have a clear concurrency semantics because it affects the validity of basic optimizations.

$$\frac{\begin{array}{c} A \subseteq G.\mathsf{E}_{e.\mathsf{tid}} \qquad \mathsf{dom}([\mathsf{E}_{e.\mathsf{tid}}]\ ;\ \mathsf{po}\ ;\ [A]) \subseteq A \qquad \mathit{labels}(\mathsf{sequence}_{\mathsf{po}}(A)) \cdot e.\mathsf{lab} \in \mathbb{P}(e.\mathsf{tid}) \\ \mathsf{E}' = \mathsf{E} \uplus \{e\} \\ \mathsf{po}' = \mathsf{po} \cup (A \times \{e\}) \qquad \mathsf{isCons}_M(\langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle) \qquad CF = (\mathsf{E}_{e.\mathsf{tid}} \setminus A) \\ \textbf{if } e \in \mathcal{R} \textbf{ then } \exists w \in \mathsf{E} \cap \mathcal{W}.\ \mathsf{jf}' = \mathsf{jf} \cup \{(w,e)\} \wedge w.\mathsf{loc} = e.\mathsf{loc} \wedge \\ ((w,e) \in G'.\mathsf{Race}(\mathrm{NA}) \wedge e.\mathsf{rval} = \mathbf{u} \vee w.\mathsf{wval} = e.\mathsf{rval}) \\ \textbf{else } \mathsf{jf}' = \mathsf{jf} \\ EW \subseteq \{w \in \mathcal{W} \cap CF \mid w.\mathsf{loc} = e.\mathsf{loc} \wedge w.\mathsf{wval} = e.\mathsf{wval}\} \qquad \mathsf{ew}' = \mathsf{ew} \cup (W \times \{e\})^{=} \\ W \subseteq AW = \{w \in \mathcal{W} \cap \mathsf{E} \setminus CF \mid w.\mathsf{loc} = e.\mathsf{loc} \wedge e \in \mathcal{W}\} \qquad \mathsf{mo}' = \mathsf{mo} \cup W \times \{e\} \cup \{e\} \times (AW \setminus W) \end{array}}{\langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo} \rangle \rightarrow_{\mathbb{P},M} \langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle}$$

Figure 5.4: WEAKESTMO-LLVM event structure construction rules where $G' = \langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}' \rangle$. The LLVM specific change is in green.

5.3.3 Variants of WEAKESTMO

Concerning the semantics of data races, we define two variants of the WEAKESTMO model.

WEAKESTMO-C11 According to the C and C++ standards, if a program has a consistent execution with NA-race, then the program has *undefined* behavior: i.e., the program may generate any arbitrary outcome [30, 29]. To model this semantics of races, we say that a program has arbitrary behavior if it contains an extracted execution with a NA-race. Thus the WEAKESTMO-C11 has the same set of rules as WEAKESTMO model.

WEAKESTMO-LLVM We propose WEAKESTMO-LLVM, a formalization of the LLVM semantics discussed in § 5.3.2. WEAKESTMO-LLVM is a variant of the WEAKESTMO model where we extend the event structure construction rule for the racy reads as shown in Figure 5.4. The consistency constraints, execution extraction, and consistency constraints on executions in WEAKESTMO-LLVM are same as those of WEAKESTMO model.

In the remainder of this thesis, 'WEAKESTMO' refers to both variants of the model.

Chapter Summary In this chapter we proposed the WEAKESTMO model based on event structure. We observed the subtle differences between WEAKEST and WEAKESTMO models and have demonstrated that WEAKESTMO model and PS are incomparable. Then we formally defined WEAKESTMO consistency constraints, event structure construction, and execution extraction from a constructed WEAKESTMO event structure. Then we formalize the LLVM concurrency semantics based on the WEAKESTMO model. In the next chapter, we will discuss various results on our proposed formalizations.

6 Programmability Results

In this chapter, we establish some results indicating that WEAKESTMO is a good model as far as programmers are concerned. First, we evaluate the possible outcomes of WEAKEST and WEAKESTMO on the Java causality tests [45], which is a standard set of benchmarks for evaluating language-level memory models. Second, we show that WEAKESTMO provides a few standard DRF guarantees that provide stronger semantics for programs without certain kinds of races.

6.1 Java Causality Tests

We evaluate WEAKEST and WEAKESTMO on the Java causality tests [45]. Each test consists of a program along with a particular behavior and a remark as to whether that behavior should be allowed or not. For each test, we check whether our models can yield the behavior in question.

Our two models agree with each other and with promising semantics on all the tests and with the prescribed Java behavior on 17/20 tests as shown in Chakraborty [21, Appendix B]. The allowed behaviors in the respective testcases can be explained by various transformations followed by an interleaving execution. Based on the required transformations we categorize the tests as follows.

Reordering The behaviors of testcase 7 and 11 can be explained by thread-local reordering followed by an interleaving execution. For example, consider the testcase 7 and the reorderings in both threads:

$$\begin{array}{l} 1:\ r_1 = Z; \\ 2:\ r_2 = X; \\ 3:\ Y = r_2; \end{array} \Bigg\| \begin{array}{l} 4:\ r_3 = Y; \\ 5:\ Z = r_3; \\ 6:\ X = 1; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} 2:\ r_2 = X; \\ 3:\ Y = r_2; \\ 1:\ r_1 = Z; \end{array} \Bigg\| \begin{array}{l} 6:\ X = 1; \\ 4:\ r_3 = Y; \\ 5:\ Z = r_3; \end{array}$$

The transformed program can have an outcome $r_1 = r_2 = r_3 = 1$ in an interleaving 6, 2, 3, 4, 5, 1 where the reads read the concurrently writes. The respective event structure is in Figure 6.1.

False Dependence Elimination The testcases 2, 3, 6 contain false dependencies. A compiler may remove these dependencies by intrathread analyses and then perform reordering transformations. The transformed program can have an interleaving execution which leads to

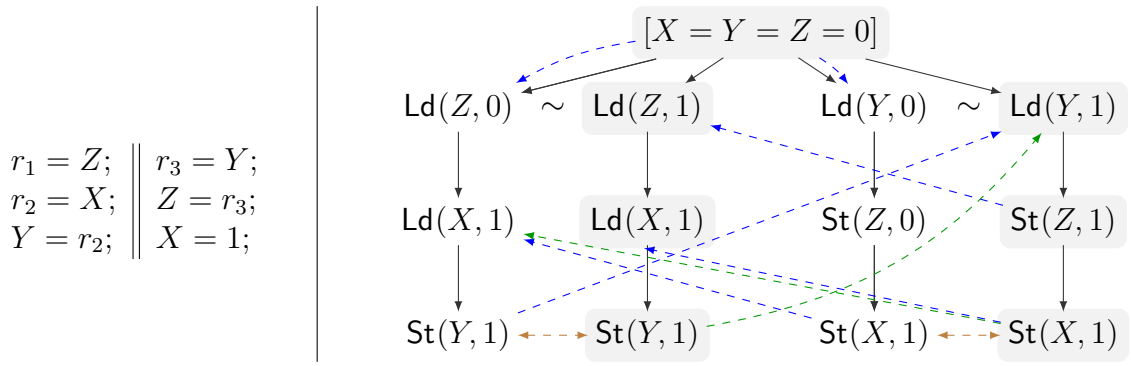


Figure 6.1: Event structure for behavior $r_1 == r_2 == r_3 == 1$ in test case 7.

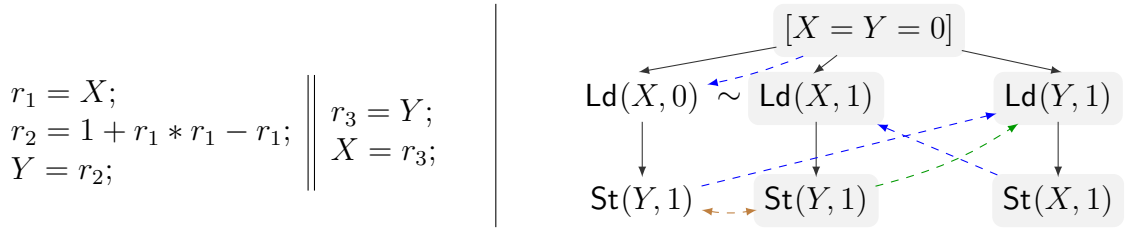


Figure 6.2: Event structure for behavior $r_1 == r_2 == 1$ in test case 8.

respective behavior. Consider the transformations on testcase 2.

$$\begin{array}{l} 1: r_1 = X; \\ 2: r_2 = X; \\ 3: \mathsf{if}(r_1 == r_2) \\ 4: \quad Y = 1; \end{array} \rightsquigarrow \begin{array}{l} 1: r_1 = X; \\ 2': r_2 = r_1; \\ 4: Y = 1; \end{array} \rightsquigarrow \begin{array}{l} 4: Y = 1; \\ 1: r_1 = X; \\ 2': r_2 = r_1; \end{array} \quad \textbf{Context:} \left[\begin{array}{l} - \\ \| \\ 5: r_3 = Y; \\ 6: X = r_3; \end{array} \right]$$

The first transformation performs read-after-read elimination followed by a false control dependence elimination. Then, the second transformation reorders the write on Y before the other instructions in the first thread. The resulting program can have an interleaving 4, 5, 6, 1, $2'$ which results in $r_1 = r_2 = r_3 = 1$ outcome. The respective event structure is in Figure 6.2.

Global Value Speculation The behaviors in testcases 1, 8, 9, 9a, 17, 18 require global value speculation analyses. Using global or inter-thread analysis, it is possible to eliminate certain dependencies.

Control Dependence Elimination In testcase 1, global value speculation can eliminate a control dependency and enable the weak outcome as follows:

$$\begin{array}{l} 1: r_1 = X; \text{ // 0 or 1} \\ 2: \mathsf{if}(r_1 \geq 0) \\ 3: \quad Y = 1; \end{array} \rightsquigarrow \begin{array}{l} 1: r_1 = X; \\ 3: Y = 1; \end{array} \rightsquigarrow \begin{array}{l} 3: Y = 1; \\ 1: r_1 = X; \end{array} \quad \textbf{Context:} \left[\begin{array}{l} - \\ \| \\ 4: r_2 = Y; \\ 5: X = r_2; \end{array} \right]$$

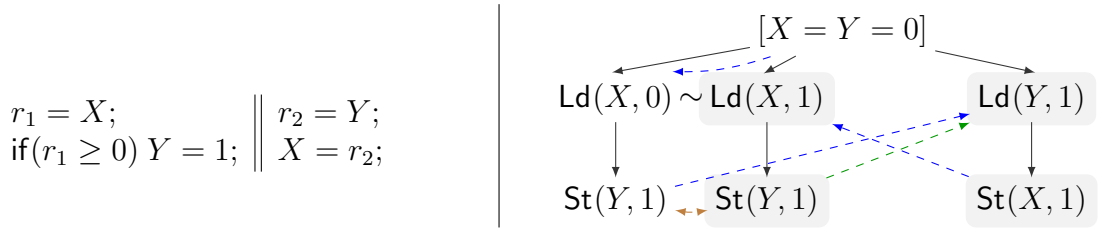


Figure 6.3: Event structure for behavior $r_1 == r_2 == 1$ in test case 1.

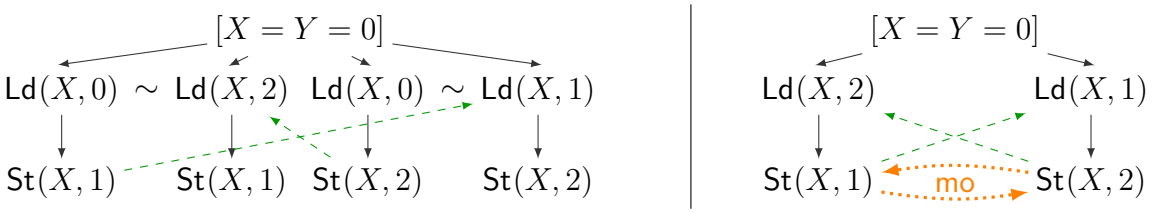
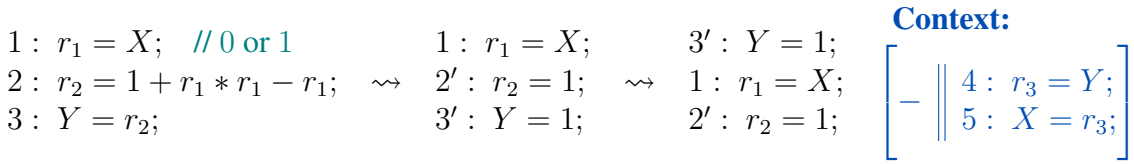


Figure 6.4: Event structure for test case 16 forbids $r_1 = 2 \wedge r_2 = 1$.

Global value analysis can identifies that r_1 is either 0 or 1, which makes the condition on line 2 always true, thereby removing the control dependency between load of X and store of Y. Then reordering the store of Y before the load of X, and considering the interleaving 3, 4, 5, 1 results in the outcome $r_1 = r_2 = 1$. The event structure capturing the behavior is shown in Figure 6.3.

Data Dependence Elimination The behaviors in testcases 8, 9, and 9a result from the eliminations of data dependencies using global value speculation analyses. For example, consider testcase 8:



Global value analysis identifies that X and Y can have the values 0 or 1. Thus, r_1 reads either 0 or 1, and so r_2 and in consequence Y is always 1. The first transformation then removes the dependency from instruction 1 to 3, which enables them to be reordered in the second transformation. The target program can execute the instructions in the following order: $3'$, 4, 5, 1, $2'$. This execution order results in the outcome $r_1 = r_2 = 1$.

The testcases 9 and 9a are similar to testcase 8 and are discussed in Manson et al. [45]. Transformations based on global value analysis can similarly eliminate data dependencies in testcases 17 and 18 to yield the respective outcomes.

Exceptions Among the remaining cases, our model (as well as promising semantics [33]) do not agree to Manson et al. [45] regarding the outcomes of certain tests.

Coherence Our proposed models disagree with the outcome in test case 16:

$$\begin{array}{l} r_1 = X; \quad \text{// 2} \\ X = 1; \end{array} \Bigg\| \begin{array}{l} r_2 = X; \quad \text{// 1} \\ X = 2; \end{array}$$

The behavior $r_1 = 2 \wedge r_2 = 1$ of test 16 is a classic coherence violation: while Java allows this outcome, promising semantics [33] as well as our models forbid it. As shown in Figure 6.4, WEAKEST event structure accommodate $\mathsf{Ld}(X, 2)$ and $\mathsf{Ld}(X, 1)$ events in an event structure, but disallows the behavior in an extracted execution. On the other hand, WEAKESTMO forbids the construction of any event structure which contains both $\mathsf{Ld}(X, 2)$ and $\mathsf{Ld}(X, 1)$ events.

Thread Sequentialization Tests 19 and 20 are due to thread sequentialization (i.e., $\mathbb{C}_1 \,||\, \mathbb{C}_2 \rightsquigarrow \mathbb{C}_1 \,;\, \mathbb{C}_2$) of the tests 17 and 18 respectively: Java allows the weak outcomes, whereas our models do not. Thread sequentialization is unsound in our models as shown in Figure 10.1 similar to that of promising semantics [33, §6].

In addition, we remark that the surprisingly weak outcomes of tests 5 and 10 (forbidden by our models) can also be explained by thread sequentialization. For instance, consider the following transformations on test case 10:

$$\begin{array}{l} r_2 = Y; \\ \mathsf{if}(r_2 == 1) \\ \quad X = 1; \end{array} \Bigg\| \begin{array}{l} r_3 = Z; \\ \mathsf{if}(r_3 == 1) \\ \quad X = 1; \end{array} \Bigg\| \; Z = 1; \quad \rightsquigarrow \quad \begin{array}{l} r_2 = Y; \\ \mathsf{if}(r_2 == 1)\{ \\ \quad X = 1; \\ \quad \{ \\ \quad r_3 = Z; \\ \quad \mathsf{if}(r_3 == 1) \\ \quad\quad X = 1; \\ \quad \} \\ \} \\ \mathsf{else}\{ \\ \quad X = 0; \\ \quad \{ \\ \quad r_3 = Z; \\ \quad \mathsf{if}(r_3 == 1) \\ \quad\quad X = 1; \\ \quad \} \\ \} \end{array} \Bigg\| \; Z = 1;$$

Context:

$$\left[\; - \; \Bigg\| \begin{array}{l} r_1 = X; \\ \mathsf{if}(r_1 == 1) \\ \quad Y = 1; \end{array} \right]$$

The transformation finds that the value of X is 0 or 1. Hence it introduces a dead branch with $X = 0$ when $r_2 \neq 1$. Then it sequentializes two threads in the two branches where $r_2 == 1$ and $r_2 \neq 1$. In Chakraborty [21, Appendix B.1] we discuss further transformations which results in an outcome $r_1 = r_2 = 1 \wedge r_3 = 0$.

Thus thread sequentialization along with other transformation result in a program outcome which is forbidden in WEAKESTMO, promising semantics [33], or Java models.

6.2 Data-Race-Freedom Guarantees

In this section, we show that WEAKESTMO provides certain guarantees for programs without races. These guarantees are stated as DRF ("data-race-freedom") theorems, and allow programmers to understand the behavior of a certain class of programs without needing to understand the underlying formal model.

We start with some basic definitions. An execution X is:

- *RA-race-free* if its races (cf. §5.3) are confined to SC accesses (i.e., $\mathsf{X.Race} \subseteq \mathcal{E}_{\text{SC}}$);
- *RLX-race-free* if $\mathsf{X.Race} \subseteq \mathsf{Ld}_{\sqsupseteq \text{ACQ}} \cup \mathsf{St}_{\sqsupseteq \text{REL}} \cup \mathsf{U}_{\sqsupseteq \text{ACQ-REL}}$.

By definition, an execution that is RA-race-free is also RLX-race-free.

DRF-RLX We first state and prove the DRF-RLX theorem, which provides a very basic correctness guarantee for RLX-race-free programs. It says that such programs have only $(\mathsf{po} \cup \mathsf{rf})$-acyclic executions.

Theorem 2 (DRF-RLX)**.** *Given a program $\mathbb{P}$, suppose its RC11-consistent executions are RLX-race-free. Then,* $\mathsf{Behavior}_{\text{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{RC11}(\mathbb{P})$.

To prove this theorem, we first establish a few helper lemmas. We first show that $G.\mathsf{jf} \subseteq G.\mathsf{hb}$ holds for every event structure G constructed from $\mathbb{P}$ (Lemma 3). It then follows in Lemma 5 that at most one full execution X can be extracted from G with $\mathsf{X.rf} = G.\mathsf{jf}$ (Lemma 4), and so that execution has $\mathsf{X.po} \cup \mathsf{X.rf}$ be acyclic.

Lemma 3. *Given a program $\mathbb{P}$, suppose all its* RC11*-consistent executions are* RLX*-race-free. Let G be an event structure such that $G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKESTMO}}{}^{*} G$. Then, $G.\mathsf{jf} \subseteq G.\mathsf{hb}$ holds.*

Proof. We show $G.\mathsf{jf} \subseteq G.\mathsf{hb}$ holds by induction on the construction of G. It holds trivially for $G = G_{\mathsf{init}}$ because $G_{\mathsf{init}}.\mathsf{jf} = \emptyset$.

For the inductive case, we know that $G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKESTMO}}{}^{*} G \rightarrow_{\mathbb{P},\text{WEAKESTMO}} G'$ and $G.\mathsf{jf} \subseteq G.\mathsf{hb}$, and have to show that $G'.\mathsf{jf} \subseteq G'.\mathsf{hb}$. We do case analysis on the step $G \rightarrow_{\mathbb{P},\text{WEAKESTMO}} G'$; let e be the event appended to G to construct G'.

Case $e \notin \mathcal{R}$. In this case, $G'.\mathsf{jf} = G.\mathsf{jf}$ and $G.\mathsf{hb} \subseteq G'.\mathsf{hb}$. Hence $G'.\mathsf{jf} \subseteq G'.\mathsf{hb}$ holds.

Case $e \in \mathcal{R}$. In this case, there exists a write $w \in G.\mathsf{E}$ such that $G'.\mathsf{jf} = G.\mathsf{jf} \uplus \{(w, e)\}$. We consider the following cases for $G.\mathsf{jf}(w, e)$:

Subcase $(w, e) \in G'.\mathsf{hb}$. In this case, $G'.\mathsf{jf} \subseteq G'.\mathsf{hb}$ holds.

Subcase $(e, w) \in G'.\mathsf{hb}$. This case is not possible as it violates (COH') in G'.

Subcase $(w, e) \notin G'.\mathsf{hb}^{=}$. In this case, $(w, e) \in G'.\mathsf{Race}(\text{RLX})$.

We take A to be the $G'.\mathsf{hb}$-prefixes of e and w. From (CFJ), it follows that A is conflict-free.

Let G'' be the restriction of G' to A. By construction, G'' is conflcit-free WEAKESTMO consistent event structure which is an RC11 execution and $(w, e) \in G''.\mathsf{Race}(\text{RLX})$. This contradicts the antecedent, and hence the statement holds. □

Lemma 4. *Given a program $\mathbb{P}$, suppose all its* RC11*-consistent executions are* RLX*-race-free. Let X be an execution extracted from a* WEAKESTMO *event structure G of $\mathbb{P}$ (that is, $G_{\mathsf{init}} \rightarrow_{\mathbb{P},\text{WEAKESTMO}}{}^{*} G$ and $\mathsf{X} \in \mathsf{ex}_{\text{WEAKESTMO}}(G)$). Then $\mathsf{X.rf} \subseteq G.\mathsf{jf}$ holds.*

Proof. Assume by contradition that there exist $(w_1, r) \in \mathsf{X}.\mathsf{rf} \setminus G.\mathsf{jf}$. Then, there exists w_2 such that $G.\mathsf{ew}(w_1, w_2)$ and $(w_2, r) \in G.\mathsf{jf}$. From Lemma 3, we know $(w_2, r) \in G.\mathsf{hb}$. From the definition of execution extraction, we know $w_2 \in \mathsf{X}.\mathsf{E}$. The latter contradicts that $w_1 \in \mathsf{X}.\mathsf{E}$ and hence the statement holds. □

Lemma 5. *Given a program $\mathbb{P}$, suppose all its* RC11*-consistent executions are* RLX*-race-free. Every execution extracted from a* WEAKESTMO *event structure of $\mathbb{P}$ is* RC11*-consistent.*

Proof. Let G be a WEAKESTMO event structure of $\mathbb{P}$ and X be an execution extracted from it. It suffices to show that $(\mathsf{X}.\mathsf{po} \cup \mathsf{X}.\mathsf{rf})$ is acyclic. From Lemmas 3 and 4, we know $(\mathsf{X}.\mathsf{po} \cup \mathsf{X}.\mathsf{rf}) \subseteq (G.\mathsf{po} \cup G.\mathsf{jf}) \subseteq G.\mathsf{hb}$. And since $G.\mathsf{hb}$ is acyclic, the conclusion holds. □

Proof of Theorem 2. The $\subseteq$ direction follows from Lemma 5, while the $\supseteq$ direction is trivial because RC11 is stronger than WEAKESTMO. □

Stronger DRF Results Based on the DRF-RLX theorem, we can proceed to establish stronger DRF results.

Composing our DRF-RLX theorem with the DRF-SC theorem of Lahav et al. [37, Theorem 4], we can derive a standard DRF-SC theorem, which says that programs whose races under SC are restricted to SC accesses exhibit only SC behavior.

Theorem 3 (DRF-SC)**.** *Given a program $\mathbb{P}$, suppose its SC-consistent executions are RA-race-free. Then,* $\mathsf{Behavior}_{\text{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{SC}(\mathbb{P})$.

We can further combine our DRF-RLX theorem with Lahav et al. [37, Theorem 5] to get another simple criterion for ensuring the absence of weak behaviors. Namely, if all accesses of a program $\mathbb{P}$ to shared locations are atomic (i.e., at least RLX) and shared-location accesses in the same thread are separated by an SC-fence, then that program has only SC behaviors.

Finally, composing our DRF-RLX theorem with the DRF-RA theorem of Kang et al. [33, Theorem 2], we can derive a DRF-RA theorem for WEAKESTMO. The DRF-RA theorem states that a program whose races under release-acquire consistency (i.e., by treating all accesses as having release/acquire semantics) are confined to release/acquire or SC accesses exhibits only release-acquire consistent behavior.

Theorem 4 (DRF-RA)**.** *Given a program $\mathbb{P}$, suppose its RA-consistent executions are RLX-race-free. Then,* $\mathsf{Behavior}_{\text{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{RA}(\mathbb{P})$.

These data-race-freedom properties ensure that if a program contains only stricter memory accesses then the proposed WEAKESTMO models do not show any weaker behavior and in consequence reasoning about such a program in WEAKESTMO model can be done in relatively straightforward manner.

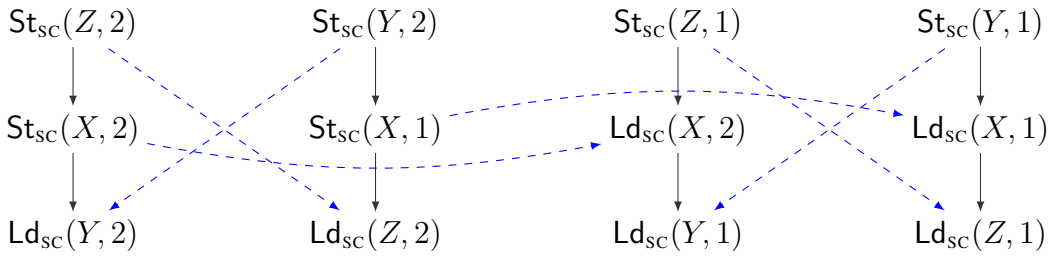


Figure 6.5: WEAKEST event structure of SC-WEAKEST program with $a = b = c = 2 \wedge d = r = s = 1$. The event structure, as an execution, is not RC11 consistent.

Data-Race-Freedom Guarantees in WEAKEST We observe that WEAKEST model is not strong enough to establish the data-race-freedom guarantee. Consider the following program adapted from Sezgin [68].

$$\begin{array}{l|l|l|l} Z_{\mathrm{sc}} = 2; & Y_{\mathrm{sc}} = 2; & Z_{\mathrm{sc}} = 1; & Y_{\mathrm{sc}} = 1; \\ X_{\mathrm{sc}} = 2; & X_{\mathrm{sc}} = 1; & c = X_{\mathrm{sc}}; \ // \ 2 & r = X_{\mathrm{sc}}; \ // \ 1 \\ a = Y_{\mathrm{sc}}; \ // \ 2 & b = Z_{\mathrm{sc}}; \ // \ 2 & d = Y_{\mathrm{sc}}; \ // \ 1 & s = Z_{\mathrm{sc}}; \ // \ 1 \end{array} \qquad \text{(SC-WEAKEST)}$$

In this program no interleaving execution results in $a = b = c = 2 \wedge d = r = s = 1$ outcome. However, the WEAKEST model allows to create an event structure as shown in Figure 6.5 which creates the respective events. The outcome is discarded due to the WeakRC11 consistency constraints on the extracted executions.

While the example does not contradict the data-race-freedom guarantees, it fails WEAKEST model to establish Lemma 3; the event structure in Figure 6.5 is conflcit-free WEAKEST consistent event structure but as an execution fails RC11 consistency constraints.

WEAKESTMO does not allow this event structure as it enforces mo either from $\mathsf{St}(X,2)$ to $\mathsf{St}(X,1)$ or vice versa and in consequence would not allow both $\mathsf{Ld}(X,2)$ and $\mathsf{Ld}(X,1)$ in a single event structure.

Chapter Summary In this chapter, we evaluated WEAKESTMO from the perspective of programmability. We showed that it produces the expected results on the Java causality tests and that it satisfies the standard DRF guarantees. In the next chapter, we will evaluate WEAKESTMO from the perspective of performance, showing how it can be used in compilation.

7 Compilation Results

In this chapter, we discuss the use of WEAKESTMO as an intermediate memory model for an optimizing compiler from C/C++ to target architectures such as x86, PowerPC, and ARM. Typically, compilation takes place in multiple steps. First, the C/C++ program is mapped to the compiler's intermediate representation (IR), which will follow the WEAKESTMO semantics. Then, a number of optimizing transformations are performed in the IR, and finally the IR is mapped to the code for the respective target architecture. We first define when a transformation is correct.

Definition 8. A transformation of program $\mathbb{P}_{\mathsf{src}}$ in memory model M_{src} to program $\mathbb{P}_{\mathsf{tgt}}$ in model M_{tgt} is *correct* if it does not introduce new behaviors:
i.e., $\mathsf{Behavior}_{M_{\mathsf{tgt}}}(\mathbb{P}_{\mathsf{tgt}}) \subseteq \mathsf{Behavior}_{M_{\mathsf{src}}}(\mathbb{P}_{\mathsf{src}})$.

We then study the correctness of the aforementioned transformation steps. The correctness proofs of these transformations are in Chakraborty and Vafeiadis [23, Appendices E to H].

7.1 Mapping from C/C++ to WEAKESTMO

WEAKESTMO supports all the features of the C11 model except for `memory_order_consume` loads. Thus, after strengthening the consume loads into acquire loads (a transformation that is meant to be sound in C11), the mapping from C11 to WEAKESTMO is simply the identity mapping.

We take the C11 concurrency model to be its revised formal definition, i.e., the weakRC11 model by Lahav et al. [37], which is the RC11 model without the $(\mathsf{po} \cup \mathsf{rf})$-acyclicity constraint. Since we use the same weakRC11 model for constraining inconsistent executions (§3.3.3), the correctness of the mapping is straightforward.

Theorem 5. *The identity mapping from* weakRC11 *to* WEAKESTMO *is correct (for both models).*

7.2 Optimizations as WEAKESTMO Source-to-Source Transformations

We move on to consider the correctness of standard compiler optimizations. Following the related work [66, 74], we consider compiler optimizations as being composed of a number of simple thread-local source-to-source transformations—introductions, reorderings, and eliminations of memory accesses—and we restrict attention to the correctness of those basic transformations.

$\downarrow a \setminus b \rightarrow$	$\mathsf{Ld}_{\sqsubseteq \mathrm{ACQ}}(\ell)$	$\mathsf{Ld}_{\mathrm{SC}}(\ell)$	$\mathsf{St}_{\mathrm{NA}}(\ell)$	$\mathsf{St}_{\mathrm{RLX}}(\ell)$	$\mathsf{St}_{\sqsupseteq \mathrm{REL}}(\ell)$	$\mathsf{U}_{\sqsubseteq \mathrm{ACQ}}(\ell)$	$\mathsf{U}_{\sqsupseteq \mathrm{REL}}(\ell)$	$\mathsf{F}_{\mathrm{ACQ}}$	$\mathsf{F}_{\mathrm{REL}}$	F_{SC}
$\mathsf{Ld}_{\mathrm{NA}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{Ld}_{\mathrm{RLX}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
$\mathsf{Ld}_{\sqsupseteq \mathrm{ACQ}}(\ell')$	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
$\mathsf{St}_{\sqsubseteq \mathrm{REL}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{St}_{\mathrm{SC}}(\ell')$	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{U}_{\sqsubseteq \mathrm{REL}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
$\mathsf{U}_{\sqsupseteq \mathrm{ACQ}}(\ell')$	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
$\mathsf{F}_{\mathrm{ACQ}}$	✗	✗	✗	✗	✗	✗	✗	=	✗	✗
$\mathsf{F}_{\mathrm{REL}}$	✓	✓	✓	✗	✓	✗	✓	✓	=	✗
F_{SC}	✗	✗	✗	✗	✗	✗	✗	✗	✗	=

Table 7.1: Allowed reorderings $a{\cdot}b \rightsquigarrow b{\cdot}a$ where $\ell \neq \ell'$.

To describe these transformations on a thread i, we use the informal syntax $\alpha \rightsquigarrow \beta$, where α and β are sequences of memory accesses possibly with metavariables ranging over both α and β. By this syntax, we mean that every trace of memory accesses by the target code $\mathbb{P}_{\mathsf{tgt}}(i)$ is also a trace of the source code of $\mathbb{P}_{\mathsf{src}}(i)$ where at most one subsequence α of the trace has been replaced with β. More formally, $\mathbb{P}_{\mathsf{tgt}}(i) \subseteq \mathbb{P}_{\mathsf{src}}(i) \cup \{\tau{\cdot}\beta{\cdot}\tau' \mid \tau{\cdot}\alpha{\cdot}\tau' \in \mathbb{P}_{\mathsf{src}}(i)\} \wedge \forall j \neq i.\ \mathbb{P}_{\mathsf{tgt}}(j) = \mathbb{P}_{\mathsf{src}}(j)$.

Before enumerating the set of verified transformations, we outline the general proof structure. Let the source program $\mathbb{P}_{\mathsf{src}}$ be transformed to the target program $\mathbb{P}_{\mathsf{tgt}}$. Given an event structure, G_{tgt}, generated by the target program (i.e., $G_{\mathsf{init}} \rightarrow_{\mathbb{P}_{\mathsf{tgt}},\mathrm{WEAKESTMO}}{}^{*}\ G_{\mathsf{tgt}}$) and an execution, $\mathsf{X}_t \in \mathsf{ex}_{\mathrm{WEAKESTMO}}(G_{\mathsf{tgt}})$, extracted from it, we construct a corresponding event structure, G_{src}, that can be generated by the source program (i.e., $G_{\mathsf{init}} \rightarrow_{\mathbb{P}_{\mathsf{src}},\mathrm{WEAKESTMO}}{}^{*}\ G_{\mathsf{src}}$) and an execution, $\mathsf{X}_s \in \mathsf{ex}_{\mathrm{WEAKESTMO}}(G_{\mathsf{src}})$, that is extracted from it. We then show that the execution X_s is consistent, that it has the same outcomes as X_t (i.e., $\mathsf{Behavior}(\mathsf{X}_t) = \mathsf{Behavior}(\mathsf{X}_s)$), and that if X_t has a non-atomic race, then so does X_s. Formally, the last requirement is that $\mathsf{X}_t.\mathsf{Race} \cap \mathcal{E}_{\mathrm{NA}} \neq \emptyset \implies \mathsf{X}_s.\mathsf{Race} \cap \mathcal{E}_{\mathrm{NA}} \neq \emptyset$.

Reordering of Independent Accesses Compilers often reorder a pair of independent instructions as part of instruction scheduling or in order to enable further optimizations.

The safe (✓) and unsafe (✗) reorderings are listed in Table 7.1. We prove the correctness of the safe transformations by constructing for each target execution a source execution with the same behavior.

Theorem 6. *The safe reorderings in Table 7.1 are correct in both* WEAKESTMO *models.*

The proof of this theorem follows the proof structure just mentioned. The proof details are presented in Chakraborty [21, Appendix F].

Counterexamples for the unsafe reorderings can be found in Vafeiadis et al. [74]. As a side remark, GCC and LLVM do not exploit the full range of correct reorderings: they typically reorder only non-atomic accesses with respect to other accesses, but do not reorder pairs of atomic accesses.

$$\mathsf{St}_{o'}(x,v')\cdot\mathsf{St}_{o}(x,v) \rightsquigarrow \mathsf{St}_{o}(x,v) \quad \text{(Overwritten write)}$$
$$\mathsf{St}_{o}(x,v)\cdot\mathsf{Ld}_{o'}(x,v) \rightsquigarrow \mathsf{St}_{o}(x,v) \quad \text{(Read after write)}$$
$$\mathsf{U}_{o}(x,v',v)\cdot\mathsf{Ld}_{o'}(x,v) \rightsquigarrow \mathsf{U}_{o}(x,v',v) \quad \text{(Read after update)}$$
$$\mathsf{Ld}_{o}(x,v)\cdot\mathsf{Ld}_{o'}(x,v) \rightsquigarrow \mathsf{Ld}_{o}(x,v) \quad \text{(Read after read)}$$
$$\mathsf{Ld}_{\text{NA}}(x,v)\cdot\mathsf{St}_{\text{NA}}(x,v) \rightsquigarrow \epsilon \quad \text{(Non-atomic read write)}$$
$$\mathsf{St}_{\text{NA}}(x,v')\cdot\tau\cdot\mathsf{St}_{\text{NA}}(x,v) \rightsquigarrow \tau\cdot\mathsf{St}_{\text{NA}}(x,v) \quad \text{(Non-adjacent overwritten write)}$$
$$\mathsf{St}_{\text{NA}}(x,v)\cdot\tau\cdot\mathsf{Ld}_{\text{NA}}(x,v) \rightsquigarrow \mathsf{St}_{\text{NA}}(x,v)\cdot\tau \quad \text{(Non-adjacent read after write)}$$

Figure 7.1: Safe eliminations where $o' \sqsubseteq o$ and τ does not contain any x-accesses nor any release-acquire pairs. That is, for all τ_1, τ_2 such that $\tau = \tau_1\cdot\tau_2$, either τ_1 does not contain a release label or τ_2 does not contain an acquire label.

Redundant Access Elimination We enlist a number of correct elimination transformations in Figure 7.1. The first four transformations remove an access because of an adjacent access to the same location. Compilers like GCC and LLVM perform such transformations only when the eliminated access is non-atomic (i.e., $o' = \text{NA}$); the eliminations are, however, also correct even for atomic accesses as long as the memory order of the eliminated access is not stronger than that of the justifying access (i.e., $o' \sqsubseteq o$). Combining the adjacent access eliminations with a number of safe reordering steps enables us to also eliminate certain non-adjacent accesses. Next, the fifth transformation removes a non-atomic load-store pair corresponding to the assignment $x = x$ in a programming language. The last two transformations eliminate a non-atomic access that is redundant because of a non-atomic store to the same location.

Theorem 7. *The eliminations in Figure 7.1 are correct in both* WEAKESTMO *models.*

As an example, we show the correctness proof for the overwritten write elimination, i.e., the first transformation from Figure 7.1. The proofs of the remaining transformations are similar and are shown in Chakraborty [21, Appendix G].

Proof. Recall the relationship between the two programs for the thread i affected by the transformation:

$$\mathbb{P}_{\mathsf{tgt}}(i) \subseteq \mathbb{P}_{\mathsf{src}}(i) \cup \{\tau\cdot\mathsf{St}_{o}(x,v)\cdot\tau' \mid \tau\cdot\mathsf{St}_{o'}(x,v')\cdot\mathsf{St}_{o}(x,v)\cdot\tau' \in \mathbb{P}_{\mathsf{src}}(i) \wedge o' \sqsubseteq o\}$$

For all other threads $j \neq i$, we have $\mathbb{P}_{\mathsf{tgt}}(j) = \mathbb{P}_{\mathsf{src}}(j)$. Assume we have a target event structure, G_{tgt}, and an execution, $\mathsf{X}_t \in \mathsf{ex}_{\text{WEAKESTMO}}(G_{\mathsf{tgt}})$, extracted from it.

Let W be the set of stores of thread i of G_{tgt} with label $\mathsf{St}_o(x,v)$, and whose po-prefix has some sequence of labels τ such that $\tau\cdot\mathsf{St}_o(x,v) \notin \mathbb{P}_{\mathsf{src}}(i)$. Then, because of the relationship between the two programs, we know that for each such $w \in W$, $\tau\cdot\mathsf{St}_{o'}(x,v')\cdot\mathsf{St}_o(x,v) \in \mathbb{P}_{\mathsf{src}}(i)$ for the appropriate τ. Let C be the immediate $G_{\mathsf{tgt}}.\mathsf{po}$-predecessors of the events in W.

Source Event Structure Construction. To construct G_{src}, we follow the construction steps of G_{tgt}. For each target construction step that adds event e to G_{tgt} to get G'_{tgt}, we perform one or more corresponding steps going from G_{src} to G'_{src}. We do a case analysis on the event e of the target event structure.

Case $e \notin W$**:** In this case, we append event e to the source event structure as follows:

$$
\begin{aligned}
G'_{\mathsf{src}}.\mathsf{E} &= G_{\mathsf{src}}.\mathsf{E} \uplus \{e\} \\
G'_{\mathsf{src}}.\mathsf{po} &= (G_{\mathsf{src}}.\mathsf{po} \uplus \{(a, e) \mid a \in \mathsf{dom}(G'_{\mathsf{tgt}}.\mathsf{po}; [e])\})^+ \\
G'_{\mathsf{src}}.\mathsf{jf} &= G'_{\mathsf{tgt}}.\mathsf{jf} \\
G'_{\mathsf{src}}.\mathsf{mo} &= G'_{\mathsf{tgt}}.\mathsf{mo} \cup \mathsf{imm}(G_{\mathsf{src}}.\mathsf{po}); [W]; G'_{\mathsf{tgt}}.\mathsf{mo} \cup G'_{\mathsf{tgt}}.\mathsf{mo}; [W]; \mathsf{imm}(G_{\mathsf{src}}.\mathsf{po}^{-1}) \\
G'_{\mathsf{src}}.\mathsf{ew} &= G'_{\mathsf{tgt}}.\mathsf{ew}
\end{aligned}
$$

Now we check the consistency of G'_{src}. We already know that G_{src} and G'_{tgt} are consistent. Following the construction of G'_{src}, the (CF), (CFJ), (VISJ), (ICF), (ICFJ) constraints immediately hold. It remains to show that G'_{src} satisfies (COH′).

From the definition, there is no $G_{\mathsf{src}}.\mathsf{hb}; G_{\mathsf{src}}.\mathsf{eco}^?$ as well as $G'_{\mathsf{tgt}}.\mathsf{hb}; G'_{\mathsf{tgt}}.\mathsf{eco}^?$ cycle. Compared to G_{src} and G'_{tgt}, the additional $G'_{\mathsf{src}}.\mathsf{mo}$ edges are from and to events the deleted events.

Let $d \in (G'_{\mathsf{src}}.\mathsf{E} \backslash G'_{\mathsf{tgt}}.\mathsf{E})$ be such a deleted event. Assume the mo edges to or from d creates a $G'_{\mathsf{src}}.\mathsf{hb}; G'_{\mathsf{src}}.\mathsf{eco}^?$ cycle. However, for each $G'_{\mathsf{src}}.\mathsf{mo}(d, e)$ or $G'_{\mathsf{src}}.\mathsf{mo}(e, d)$ already there exists $G'_{\mathsf{src}}.\mathsf{mo}(w, e)$ or $G'_{\mathsf{src}}.\mathsf{mo}(e, w)$ respectively where $w \in W$ and $\mathsf{imm}(G_{\mathsf{src}}.\mathsf{po}(d, w))$. Thus event e results no new $G'_{\mathsf{src}}.\mathsf{hb}; G'_{\mathsf{src}}.\mathsf{eco}^?$ cycle and hence G'_{src} satisfies (COH′).

We know G_{src} preserves (NCFU) and (NCFSC). Consider G'_{src} violates (NCFU) or (NCFSC). In that case G'_{src} violates (NCFU) or (NCFSC) due to e. However, following the construction of G'_{src}, in this case, G'_{tgt} also violates (NCFU) or (NCFSC). This is not possible as G'_{tgt} is consistent. Hence a contradiction and G'_{src} preserves (NCFU) and (NCFSC).

Case $e \in W$**:** In this case, we first append a new event d with $d.\mathsf{lab} = \mathsf{St}_{o'}(x, v')$ and then the event e to G_{src} as follows:

$$
\begin{aligned}
G'_{\mathsf{src}}.\mathsf{E} &= G_{\mathsf{src}}.\mathsf{E} \uplus \{d, e\} \quad \text{where } d.\mathsf{lab} = \mathsf{St}_{o'}(x, v') \\
G'_{\mathsf{src}}.\mathsf{po} &= (G_{\mathsf{src}}.\mathsf{po} \uplus \{(d, e)\} \uplus \{(c, d) \mid (c, e) \in G'_{\mathsf{tgt}}.\mathsf{po}\})^+ \\
G'_{\mathsf{src}}.\mathsf{jf} &= G'_{\mathsf{tgt}}.\mathsf{jf} \\
G'_{\mathsf{src}}.\mathsf{mo} &= G'_{\mathsf{tgt}}.\mathsf{mo} \uplus \{(d, a) \mid G'_{\mathsf{tgt}}.\mathsf{mo}(e, a)\} \uplus \{(a, d) \mid G'_{\mathsf{tgt}}.\mathsf{mo}(a, e)\} \uplus \{(d, e)\} \\
G'_{\mathsf{src}}.\mathsf{ew} &= G'_{\mathsf{tgt}}.\mathsf{ew}
\end{aligned}
$$

Now we check the consistency of G'_{src}. We already know that G_{src} and G'_{tgt} is consistent. Following the construction of G'_{src}, the (CF), (CFJ), (VISJ), (ICF), (ICFJ) constraints immediately hold. It remains to show that G'_{src} satisfies (COH′).

From the definition, there is no $G_{\mathsf{src}}.\mathsf{hb}; G_{\mathsf{src}}.\mathsf{eco}^?$ as well as $G'_{\mathsf{tgt}}.\mathsf{hb}; G'_{\mathsf{tgt}}.\mathsf{eco}^?$ cycle. Compared to G_{src} and G'_{tgt}, the additional $G'_{\mathsf{src}}.\mathsf{mo}$ edges are from and to the event d. Assume the mo edges to or from d creates a $G'_{\mathsf{src}}.\mathsf{hb}; G'_{\mathsf{src}}.\mathsf{eco}^?$ cycle.

However, for each $G'_{\mathsf{src}}.\mathsf{mo}(d,a)$ or $G'_{\mathsf{src}}.\mathsf{mo}(a,d)$ already there exists $G'_{\mathsf{src}}.\mathsf{mo}(w,e)$ or $G'_{\mathsf{src}}.\mathsf{mo}(e,w)$ respectively where $a \neq e$. Thus event e results no new $G'_{\mathsf{src}}.\mathsf{hb}; G'_{\mathsf{src}}.\mathsf{eco}^?$ cycle and hence G'_{src} satisfies (COH′).

We know G_{src} preserves (NCFU) and (NCFSC). Consider G'_{src} violates (NCFU) or (NCFSC). In that case G'_{src} violates (NCFU) or (NCFSC) due to d or e. However, following the construction of G'_{src}, in this case, G'_{tgt} also violates (NCFU) or (NCFSC). This is not possible as G'_{tgt} is consistent. Hence a contradiction and G'_{src} preserves (NCFU) and (NCFSC).

Source Execution Construction. Next, we construct an execution $\mathsf{X}_t \in \mathsf{ex}_{\text{WEAKESTMO}}(G_{\mathsf{tgt}})$.

If $W \subseteq (G_{\mathsf{tgt}}.\mathsf{E} \setminus \mathsf{X}_t.\mathsf{E})$, then we find the corresponding execution $\mathsf{X}_s \in \mathsf{ex}_{\text{WEAKESTMO}}(G_{\mathsf{src}})$ such that X_s contains no event created for $\mathsf{St}_{o'}(x,v')$. Else if an event $w_t \in W$ is in X_t, then we know that we can find an execution with $w_s \in \mathsf{X}_s.\mathsf{E}$ and $\mathsf{X}_s.\mathsf{E}$ also contains an event w' corresponding to $\mathsf{store}_{o'}(x,v')$. Thus X_s is as follows.

$$\mathsf{X}_s.\mathsf{E} = \mathsf{X}_t.\mathsf{E} \uplus \{d \mid \mathsf{X}_t.\mathsf{E} \cap W \neq \emptyset\}$$

$$\mathsf{X}_s.\mathsf{po} = (\mathsf{X}_t.\mathsf{po} \uplus \{(c,d),(d,w) \mid (c,w) \in \mathsf{imm}(\mathsf{X}_t.\mathsf{po}) \cap (C \times W) \wedge d \in (G_{\mathsf{src}}.\mathsf{E} \setminus G_{\mathsf{tgt}}.\mathsf{E})\})^+$$

$$\mathsf{X}_s.\mathsf{rf} = \mathsf{X}_t.\mathsf{rf}$$

$$\begin{aligned}\mathsf{X}_s.\mathsf{mo} = \mathsf{X}_t.\mathsf{mo} &\uplus \{(d,w) \mid (d,w) \in ((G_{\mathsf{src}}.\mathsf{E} \setminus G_{\mathsf{tgt}}.\mathsf{E}) \times W)\} \\ &\uplus \{(a,d) \mid \mathsf{X}_t.\mathsf{mo}(a,w) \wedge (d,w) \in ((G_{\mathsf{src}}.\mathsf{E} \setminus G_{\mathsf{tgt}}.\mathsf{E}) \times W) \cap \mathsf{imm}(G_{\mathsf{src}}.\mathsf{po})\} \\ &\uplus \{(d,a) \mid \mathsf{X}_t.\mathsf{mo}(w,a) \wedge (d,w) \in ((G_{\mathsf{src}}.\mathsf{E} \setminus G_{\mathsf{tgt}}.\mathsf{E}) \times W) \cap \mathsf{imm}(G_{\mathsf{src}}.\mathsf{po})\}\end{aligned}$$

Source Execution Consistency. Now we check the consistency of X_s.

Since X_t is consistent, the (Well-formed), (total-MO), (Coherence), (Atomicity) constraints also hold for X_s. The (SC) constraint is affected only when $o = o' = \text{SC}$, in which case the new events introduce some $[\mathsf{SC}], \mathsf{X}_s.\mathsf{po}_x; [\mathsf{SC}]$ edges. These edges, however, can create a $(\mathsf{X}_s.\mathsf{psc}_{\mathsf{base}} \cup \mathsf{X}_s.\mathsf{psc}_{\mathsf{F}})$ cycle only when there is a $(\mathsf{X}_t.\mathsf{psc}_{\mathsf{base}} \cup \mathsf{X}_t.\mathsf{psc}_{\mathsf{F}})$ cycle. Since X_t is consistent there is no $(\mathsf{X}_t.\mathsf{psc}_{\mathsf{base}} \cup \mathsf{X}_t.\mathsf{psc}_{\mathsf{F}})$ cycle. Hence, X_s satisfies (SC) and, as a result, X_s is consistent.

Same Behavior. For locations $y \neq x$, we have $\mathsf{X}_s.\mathsf{E}_y = \mathsf{X}.\mathsf{E}_y$ and so $\mathsf{Behavior}(\mathsf{X}_s)|_y = \mathsf{Behavior}(\mathsf{X}_t)|_y$ trivially holds. Now we check whether $\mathsf{Behavior}(\mathsf{X}_s)|_x = \mathsf{Behavior}(\mathsf{X}_t)|_x$ holds. Note that any newly introduced event $d \in \mathsf{X}_s.\mathsf{E} \setminus \mathsf{X}_t.\mathsf{E}$ is not $\mathsf{X}_s.\mathsf{mo}$ maximal, because in that case there exists $w \in W$ such that $\mathsf{X}_s.\mathsf{mo}(d,w)$. Hence $\mathsf{Behavior}(\mathsf{X}_s) = \mathsf{Behavior}(\mathsf{X}_t)$ holds.

Race Preservation. Moreover, if X_t is racy, then the new write d does not introduce any $\mathsf{X}_s.\mathsf{sw}_{\mathsf{C11}}$ edge in X_s. Hence X_s is also racy. As a result, if the target execution has undefined behavior due to a data race, so does the source execution. □

Speculative Load Introduction As we have seen in § 5.3.2, one subtle transformation that the LLVM compiler performs is speculative load introduction. This happens in cases such as the following

$$\textbf{if}\,(cond)\ \ a = X_o; \quad \rightsquigarrow \quad t = X_o; \textbf{if}\,(cond)\ \ a = t;$$

WEAKESTMO	via RC11 [37]			SPower
	x86	PowerPC	ARMv7	
$\mathsf{store}_{\sqsubseteq \mathrm{RLX}}$	mov	st	st	st
$\mathsf{store}_{\mathrm{REL}}$	mov	lwsync; st	dmb; st	lwsync; st
$\mathsf{store}_{\mathrm{SC}}$	mov; mfence	hwsync; st	dmb; st	hwsync; st
$\mathsf{load}_{\mathrm{NA}}$	mov	ld	ld	ld
$\mathsf{load}_{\mathrm{RLX}}$	mov	ld; cmp; bc	ld; cmp; bc	ld
$\mathsf{load}_{\mathrm{ACQ}}$	mov	ld; cbisync	ld; cbisb	ld; lwsync
$\mathsf{load}_{\mathrm{SC}}$	mov	hwsync; ld; cbisync	dmb; ld; cbisb	hwsync; ld; lwsync
$\mathsf{CAS}_{\mathrm{RLX}}$	lock cmpxchg	U	U	U
$\mathsf{CAS}_{\mathrm{REL}}$	lock cmpxchg	lwsync; U	dmb; U	lwsync; U
$\mathsf{CAS}_{\mathrm{ACQ}}$	lock cmpxchg	U; cbisync	U; cbisb	U, lwsync
$\mathsf{CAS}_{\mathrm{ACQ\text{-}REL}}$	lock cmpxchg	lwsync; U; cbisync	dmb; U; cbisb	lwsync; U; lwsync
$\mathsf{CAS}_{\mathrm{SC}}$	lock cmpxchg	hwsync; U; cbisync	dmb; U; cbisb	hwsync; U; lwsync
$\mathsf{F}_{\sqsubset \mathrm{SC}}$	mfence	lwsync	dmb	lwsync
F_{SC}	mfence	hwsync	dmb	hwsync

PowerPC $U \triangleq L$: lwarx; cmpw; bne L'; stwcx.; bne L; L': cbisync $\triangleq$ cmp; bc; isync

ARMv7 $U \triangleq L$: ldrex; mov; teq L'; strexeq; teq L; L': cbisb $\triangleq$ cmp; bc; isb

Figure 7.2: Compilation schemes to x86, PowerPC, and ARM.

where a conditional shared memory load is hoisted outside the conditional by introducing a fresh temporary variable, which is used only when the condition holds. As the conditional move between registers can be performed without a conditional branch, LLVM's transformation avoids introducing a branch instruction at the cost of possibly performing an unnecessary load (if the condition is false).

This transformation is incorrect in the WEAKESTMO-C11 model because it may introduce a data race in the program when $cond$ is false. It is, however, correct in WEAKESTMO-LLVM model. The proof is described in Chakraborty [21, Appendix H].

Theorem 8. *The transformation* $\epsilon \rightsquigarrow \mathsf{Ld}_o(x, \_)$ *is correct in* WEAKESTMO-LLVM.

Strengthening Finally, it is sound to strengthen the memory order of an access or fence (e.g., $\mathsf{F}_o \rightsquigarrow \mathsf{F}_{o'}$ for $o \sqsubset o'$) as well as to introduce a new fence instruction in a program (i.e., $\epsilon \rightsquigarrow \mathsf{F}_o$). These transformations are correct because WEAKESTMO is monotone.

7.3 Mapping from WEAKESTMO to x86, PowerPC, and ARMv7

Finally, we present some results about the lowering transformations from WEAKESTMO to the x86, PowerPC, and ARMv7 architectures.

First, by observing that WEAKESTMO is weaker than RC11, we use the RC11 mapping correctness results of Lahav et al. [37] to get correct mappings to the aforementioned architectures (see Figure 7.2). While these mappings are correct, they are suboptimal for PowerPC and ARMv7 in that they insert a fake conditional branch after each relaxed load.

To resolve this problem, we also prove the correctness of the intended mapping to the SPower model, a stronger version of the Power model due to Lahav and Vafeiadis [35]. In addition to the Power model's constraints, SPower requires $(\mathsf{po} \cup \mathsf{rf})$ to be acyclic, which makes the fake control dependencies after relaxed loads unnecessary. The interest of the compilation scheme to SPower is twofold. First, even though load-store reordering is architecturally allowed by the PowerPC model, as far as we know, no existing implementations actually perform such reorderings, and hence SPower is a correct description of existing implementations (at the time of writing). Second, Lahav and Vafeiadis [35] show that excluding CAS and SC accesses, PowerPC is equivalent to a model that first transforms the program by arbitrarily reordering independent memory accesses and then returns examines the behaviors of the transformed program according the SPower model. Thus, following the proof strategy of Kang et al. [33], since WEAKESTMO supports reorderings of independent memory accesses, to prove correctness of compilation to PowerPC, it suffices to prove correctness of compilation to SPower.

Proving the correctness of the intended optimal mappings from WEAKESTMO to the full PowerPC model and to ARMv7, as well as to the ARMv8-Flat model [61], is left for future work. To carry out these proofs, we intend to compile via the recent IMM model of Podkopaev et al. [59], from which compilation results to multiple architectures have been developed.

Chapter Summary In this chapter, we discussed the use of WEAKESTMO in compilation, and established the correctness of a number of low-level compiler optimizations. In the upcoming chapter, we will use these results to validate the transformations in LLVM's 'opt' phase.

8 Validating LLVM Optimizations

In the earlier chapter we have proved the correctness of safe transformations in C11 and LLVM relaxed memory concurrency and provided counter-examples for the unsafe transformations. In this chapter we check whether LLVM compiler indeed perform only the safe transformations in order to preserve compilation correctness. We present a validator for checking the correctness of LLVM 'opt' phase optimizations on C11 concurrent programs. Our validator checks that optimizations do not change memory accesses in ways disallowed by the C11 and/or LLVM memory models. We use a custom C11 concurrent program generator to trigger multiple LLVM optimizations and evaluate the efficacy of our validator. Our experiments uncovered a number of previously unknown compilation errors in the LLVM optimizations involving the C11 concurrency primitives due to the difference between the C11 and LLVM memory models.

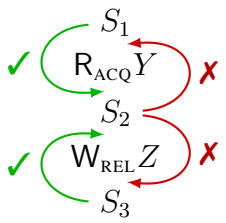
8.1 Main Ideas

We start with a very simple example as shown in Figure 8.1. The program before the transformation always uses the shared variable `g` while the lock is held, and so the two accesses to `g` are ordered. After reordering the `g = 42;` and the `unlock();` statements, however, the store to `g` is no longer protected by the lock and hence may race with the load on `g` in the second thread. While reordering `g = 42;` and `unlock();` is correct for sequential programs, it is clearly wrong for concurrent programs because it introduces a data race and, as such, it is forbidden by the C/C++11 standards [30, 29]. Essentially in the concurrent setting, we must ensure that no accesses are moved out of critical regions as in Figure 8.1. The notions of *acquire* and *release* actions generalize the notions of acquiring a lock and releasing a lock [74] as explained in Figure 8.1.

Because correctness of compiler optimizations under concurrency is still not very well understood and is an active research problem, existing compilers are typically very conservative when encountering concurrency features and often miss optimization opportunities. Consider the following C++ code snippet, where `X` is an atomic variable and two consecutive stores of order `std::memory_order_relaxed` (`RLX`) are performed.

$$\begin{array}{l} X_{\text{RLX}} = 1; \\ X_{\text{RLX}} = 3; \end{array} \quad \rightsquigarrow \quad X_{\text{RLX}} = 3;$$

Although deleting the earlier store is correct [74], both GCC and LLVM do not currently do so. Nevertheless, despite being conservative, compilers do have concurrency bugs which we discuss now.




```
atomic_int lck = 0; int g = 0;
lock() {...}
unlock() {lck = 0;}

lock();      ||  lock();      ⇝  lock();      ||  lock();
g = 42;      ||  r = g;          unlock();   ||  r = g;
unlock();    ||  unlock();       g = 42;     ||  unlock();
```

Figure 8.1: The "roach motel" principle says that shared memory accesses can be moved in critical regions but not out of them [46]. The analogue is that cockroaches (resp., accesses) can check in the motel (resp., the region), but not check out. Unsafe reordering violates "roach motel" principle and thus introduces a data race on g.

LLVM Bugs

Our validator has identified three previously unknown concurrency optimization errors in the LLVM *opt* phase:

(#22514) A combination of two *opt* transformations moves an access outside of a critical region similar to Figure 8.1.

(#22708) The "Global Value Numbering" (GVN) optimization performs an unsafe memory access reordering.

(#22306) The SLP vectorizer violates reordering constraints.

LLVM bug (#22306) The source program is race-free when *flag* is false. This is because the only access of g in the second thread happens after the SC-read of X, that synchronizes with the SC-write of X in the first thread. Moreover, one can see that the only possible result for r_2 is the value 4.

Now consider a sequence of two transformations. The first transformation moves the access of g before the conditional. From the semantics of data race in LLVM we already know that this transformation is allowed in LLVM model.

The second transformation is *repeated read elimination*, a simple special case of common subexpression elimination (CSE). Since g has been read before the X_{SC} access, it need not be read again. This transformation is valid under the C11 model, and is explicitly permitted by Sevcík [66] and Morisset et al. [48], but it is incompatible with the previous transformation. The resulting program after the two transformations is not only racy, but may return $r_2 = 0$ even under SC.

As a result of reporting this bug, the LLVM developers decided to restrict the second transformation rather than the first one, which means that the intended LLVM memory model is subtly different from the C11 model. In LLVM's model, read-write races are allowed, the non-atomic read returns an *undef* value. A write-write race, however, still results in undefined behavior.

As we have seen, this has implications on the set of allowed program transformations in the LLVM model. On the one hand, the compiler may introduce unused speculative loads as in transformation (1). On the other hand, it cannot eliminate the 'redundant' non-atomic loads

$$
\begin{array}{l||l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
g_{\text{NA}} = 4; & r_1 = 0; \\
X_{\text{SC}} = 1; & \mathsf{if}(\mathit{flag}) \\
 & \quad r_1 = g_{\text{NA}}; \\
 & \mathsf{if}(X_{\text{SC}} == 1) \\
 & \quad r_2 = g_{\text{NA}}; \\
 & \mathsf{else}\ r_2 = 4;
\end{array}
\quad \overset{(1)}{\rightsquigarrow} \quad
\begin{array}{l||l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
g_{\text{NA}} = 4; & r_1 = 0; \\
X_{\text{SC}} = 1; & t_1 = g_{\text{NA}}; \quad \textit{// introduced} \\
 & r_1 = (\mathit{flag})?\ t_1 : 0; \\
 & t_2 = X_{\text{SC}}; \\
 & t_3 = g_{\text{NA}}; \\
 & r_2 = (t_2 == 1)?\ t_3 : 4;
\end{array}
$$

(1) introduces a speculative read of g during CFG simplification.

$$
\begin{array}{l||l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
g_{\text{NA}} = 4; & r_1 = 0; \\
X_{\text{SC}} = 1; & t_1 = g_{\text{NA}}; \\
 & r_1 = (\mathit{flag})?\ t_1 : 0; \\
 & t_2 = X_{\text{SC}}; \\
 & t_3 = g_{\text{NA}}; \\
 & r_2 = (t_2 == 1)?\ t_3 : 4;
\end{array}
\quad \overset{(2)}{\rightsquigarrow} \quad
\begin{array}{l||l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
g_{\text{NA}} = 4; & r_1 = 0; \\
X_{\text{SC}} = 1; & t_1 = g_{\text{NA}}; \\
 & r_1 = (\mathit{flag})?\ t_1 : 0; \\
 & t_2 = X_{\text{SC}}; \\
 & \textit{// deleted } t_3 = g_{\text{NA}}; \\
 & r_2 = (t_2 == 1)?\ t_1 : 4;
\end{array}
$$

(2) remove redundant read of g by the GVN pass.

Figure 8.2: A sequence of LLVM transformations. The composition violates the "roach motel" property when $\mathit{flag} = \mathit{false}$.

as can be eliminated in C11 by appealing to the data race freedom (DRF) property. Hence the read elimination rule becomes:

Read-Elimination: $a{\cdot}C{\cdot}\mathsf{Ld}_{\text{NA}}(X) \rightsquigarrow a{\cdot}C$ where a is $\mathsf{St}_{\text{NA}}(X, \_)$ or $\mathsf{Ld}_{\text{NA}}(X)$ and C does not contain an acquire label.

However, LLVM semantics allows the read-after-write elimination over an acquire access as shown in §7.2. This optimization was performed by LLVM version 3.6 but was later dropped in version 3.7 while fixing this bug. This is one instance which demonstrates that it is important for LLVM to have a clear concurrency semantics because it affects the validity of basic optimizations.

Bug #22708 The "Global Value Numbering" (GVN) pass performs the transformation shown in Figure 8.3. Assume that the code is executed in the concurrent context shown in the figure and that all variables are initialized to 0: in particular, $\mathit{flag} = \mathsf{false}$.

The source program is race-free and can return only $r' = 8$ because if the program reads $X_{\text{ACQ}} \neq 0$, then it synchronizes with the parallel thread and sees the $g_{\text{NA}} = 8$ store. The target program, however, is racy and can return $r' = 0$ even with an interleaving semantics.

Bug #22306 The "Superword-Level Parallelism" (SLP) vectorizer performs the unsafe transformation shown in Figure 8.4 unrolling the loop and combining the four g accesses.

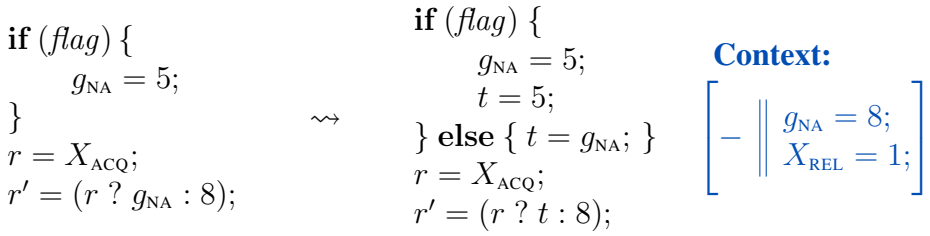


Figure 8.3: GVN performs an unsafe reordering.

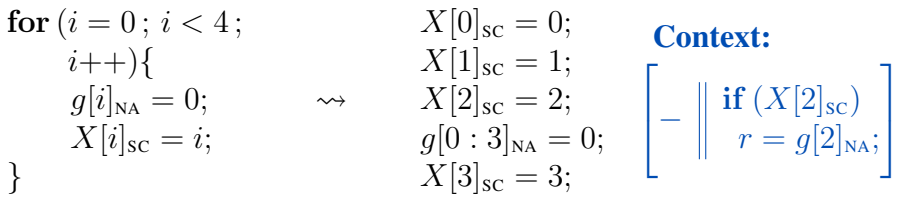


Figure 8.4: SLP vectorizer performs an unsafe reordering.

Consider running the code in the context shown in the figure with all variables initialized to 0. The source program is race-free because the $X[2]_{SC}$ accesses synchronize and therefore the $g[2]_{NA} = 0$ *happens before* the load of $g[2]_{NA}$ in the second thread. The target program, however, contains a race between the $g[2]_{NA}$ load and the $g[0:3]_{NA}$ store.

Summary We observe that all the bugs found violate the 'roach motel' principle. Bugs #22514 and #22708 reorder a load before an acquire command, whereas bug #22306 reorders a memory access after a release command. We also note that bugs #22514 and #22708 also introduced an unused load on certain paths, which is disallowed by C11 but allowed by LLVM.

8.2 Our Validation Approach

In this section, we describe our approach for validating LLVM optimizations with respect to concurrency.

The validation reads the source and the target programs and the memory model under which to perform the validation: C11 or LLVM. The validator then compares the programs by matching the shared memory accesses to identify how the transformation has affected the shared memory access sequences, and returns one of three results:

Correct: A safe matching was found between the source and the target witnessing the correctness of the transformation. This means that if we execute the target program and record the sequence τ of its memory accesses, then either the source program has undefined semantics or there is a way of executing it and obtaining a corresponding sequence σ of memory accesses that can be transformed into τ by performing the speculative load introduction, reordering and elimination rules of Table 7.1 and Figure 7.1.

Possible Error: There exists no safe match between the source and the target. We also report the cause(s) of error:

- Deletion of non-deletable accesses from the source;
- Incorrect reordering;
- Introduction of an observable write or update; or
- In case of C11, introduction of a potentially racy read operation.

Unknown: If the source program has any loop and the loop condition changes by any transformation (e.g. in loop unrolling) then the validator returns "unknown" since it does not handle such transformations.

We propose two approaches for performing the matching:

- *Compiler-independent matching (CIM).* In this scheme, the validator has no knowledge about how the memory accesses have been moved by the optimization. Thus, given the source and target access sequences, it tries to match them as precisely as possible.

- *LLVM-specific matching with instruction metadata (MD).* In this approach, we instrument the compiler so as to witness the movement of the shared memory accesses, thereby greatly simplifying the matching.

In both cases, we first map LLVM instructions to the labels defined in §3.3. In this context we term these labels as *actions*. In this mapping, we produce actions only for potentially shared accesses (i.e., accesses to global variables), not for accesses to registers or temporaries. We will now discuss the two approaches in detail.

8.2.1 Compiler Independent Matching (CIM)

In this approach, given the source and the target memory accesses we attempt to come up with a matching to check if the target is generated by a sequence of correct transformations.

We first explain how to detect if an action is redundant (§8.2.1). Then we discuss how to match the accesses on straight-line code (§8.2.1). Later we will discuss how to match programs with control flow (§8.2.1).

Marking Actions

Given the source action sequence, we categorize the actions as *non-deletable*(✓), *conditionally deletable*(⊗) or *immediately deletable*(×). Non-deletable actions are those that cannot be deleted after any set of safe reordering or deletion transformations. Conditionally deletable actions may be removed only after some other safe transformation is applied. We explain the

markings on the following sequence.

source		target
$\otimes_{\mathsf{C}_1}\ a : \mathsf{St}_{\text{RLX}}(X, \_)$		$a' : \mathsf{St}_{\text{RLX}}(X, \_)$
$\checkmark\ b : \mathsf{St}_{\text{REL}}(Y, \_)$	$\rightsquigarrow$	$c' : \mathsf{St}_{\text{RLX}}(X, \_)$
$\times\ c : \mathsf{St}_{\text{RLX}}(X, \_)$		$b' : \mathsf{St}_{\text{REL}}(Y, \_)$
$\checkmark\ d : \mathsf{St}_{\text{RLX}}(X, \_)$		$d' : \mathsf{St}_{\text{RLX}}(X, \_)$

$$\mathsf{C}_1 = [a; \mathsf{St}_{\text{RLX}}(X, \_)]$$

Actions b and d are non-deletable because they are the last writes to X and Y in the source sequence. Action c is directly deletable because it immediately precedes another similar store to X. In contrast, a is only conditionally deletable: in order to be deleted, a later relaxed store to X must be reordered before the release store to Y to satisfy the condition C_1; e.g. $b; c \rightsquigarrow c'; b'$.

Similar notions of deletable and non-deletable actions also appear in earlier work [66, 48, 74] but require adaptation to our setting of checking for the validity of an unknown sequence of transformations.

Insufficiency of Release-Acquire Pairs §7.2 introduced the lack of release-acquire pairs as a way of identifying deletable operations. We observe that presence of a release-acquire pair does not entail that an access is non-deletable. Consider the following example.

source			target		
$\otimes_{\mathsf{C}}\ a : \mathsf{St}_{\text{NA}}(g, \_)$		$\otimes_{\mathsf{C}}\ a : \mathsf{St}_{\text{NA}}(g, \_)$		$\checkmark\ c : \mathsf{Ld}_{\text{ACQ}}(Y)$	
$\checkmark\ b : \mathsf{St}_{\text{REL}}(X, \_)$	$\rightsquigarrow$	$\checkmark\ c : \mathsf{Ld}_{\text{ACQ}}(Y)$	$\rightsquigarrow$	$\times\ a : \mathsf{St}_{\text{NA}}(g, \_)$	
$\checkmark\ c : \mathsf{Ld}_{\text{ACQ}}(Y)$		$\checkmark\ b : \mathsf{St}_{\text{REL}}(X, \_)$		$\checkmark\ d : \mathsf{St}_{\text{NA}}(g, \_)$	
$\checkmark\ d : \mathsf{St}_{\text{NA}}(g, \_)$		$\checkmark\ d : \mathsf{St}_{\text{NA}}(g, \_)$		$\checkmark\ b : \mathsf{St}_{\text{REL}}(X, \_)$	

$$\mathsf{C} = [a; \mathsf{St}_{\text{NA}}(g, \_)]$$

In the source program, action a cannot be directly eliminated because there is a release-acquire pair between a and d. If, however, we transform the program by moving the acquire load earlier and/or the release store later, then in the target program, a may be removed. For this reason, we have to mark a as conditionally deletable in the source program.

C11 Release Sequences There is another subtlety in detecting deletable actions. Consider the program:

$$X_{\text{REL}} = 1; X_{\text{RLX}} = 2; X_{\text{REL}} = 3;$$

The first access to X is not deletable because according to C11, if an acquire read reads from the $X_{\text{RLX}} = 2$ store, then it synchronizes with the earlier $X_{\text{REL}} = 1$ store. Removing the $X_{\text{REL}} = 1$ store therefore removes a possible synchronization and is unsound. It is, however, conditionally deletable because if the $X_{\text{RLX}} = 2$ is deleted or strengthened to REL order, then the $X_{\text{REL}} = 1$ store can also be removed.

Synchronization Access Deletion We call *CAS*, *release*, *acquire* accesses synchronization actions. Even though according to the rules in §7.2 redundant synchronization actions may be eliminated under certain conditions, removing them goes against the programmer's intentions to communicate and synchronize across threads. Also removing such synchronization accesses can cause a deadlock or significantly degrade performance (e.g., by converting a test-and-test-and-set lock into a test-and-set lock). Moreover, currently neither LLVM nor GCC removes any atomic accesses. We therefore consider all such actions to be non-deletable.

Marking Algorithm Initially, we mark all actions to be *non-deletable*(✓) and proceed to mark individual actions as deletable(×) or conditionally deletable(⊗). For example,

- In the sequence $a : \mathsf{St}_{\text{NA}}(X, \_) \cdot C \cdot b : \mathsf{St}_{\text{NA}}(X, \_)$, a is ⊗ if C contains a release-acquire pair and deletable otherwise.
- In the sequence $a : \mathsf{Ld}_{\text{NA}}(X) \cdot C \cdot b : \mathsf{Ld}_{\text{NA}}(X)$, In C11 b is ⊗ if C contains a release-acquire pair and ✓ otherwise. In LLVM b is ⊗ if *acquire* $\in C$ and ✓ otherwise.

Matching Access Sequences

We extract the source and target actions (indexed from 1 to N) as described before, mark them as discussed in § 8.2.1, match them in multiple iterations, and finally analyze whether the matching denotes a correct transformation. We match the source and target actions in three steps:

1. **Synchronization actions.** We traverse the source and the target sequences from index 1 to N and match the synchronization actions. If any synchronization action remains unmatched or the matching is unsafe, we report "Possible Error" and return.

2. **Other non-deletable (✓) actions.** For each unmatched non-deletable source action we identify the matching window, i.e. the target subsequence within which a safe matching can occur. A matching outside the window implies that the access is unsafely reordered.

   For each non-deletable source action s, let a and b be the nearest predecessor and successor source actions such that $a; s \not\rightsquigarrow s; a$ (i.e., the $a; s \rightsquigarrow s; a$ is unsafe) and $s; b \not\rightsquigarrow b; s$ and a and b are matched with the j^{th} and k^{th} target actions respectively. The search window for s is then the target subsequence from $j + 1$ to $k - 1$. If s is a write or a release fence action then we search from the end to the start of the window and if s is a read or acquire fence action then we search from the start to the end of the window. If s remains unmatched, report "Possible Error" and return.

3. **Remaining unmatched target actions.** Since a transformation need not delete all of the deletable(×) and conditionally deletable(⊗) source actions, there may still be unmatched actions in the target sequence. To match those unmatched target actions, again we identify the appropriate search windows in the source sequence within which a safe matching may be found.

For each unmatched target action t, let a' and b' be the nearest predecessor and successor target actions such that $t; a' \not\rightsquigarrow a'; t$ and $b'; t \not\rightsquigarrow t; b'$ and a' and b' are matched to the j^{th} and k^{th} source actions respectively. The window for t is the source subsequence from $j+1$ to $k-1$. As before, for writes and release fences, we search the window from end to start, whereas for reads and acquire fences, from start to end. If t remains unmatched, we consider the access as *introduced* and analyze if the introduction of t is a safe transformation.

Note that writes and release fences are matched from end to start in both the target and the source sequences, whereas reads and acquire fences are matched from start to end. Because of this, some earlier writes and later reads may remain unmatched, but in the subsequent analysis these may be considered as redundantly introduced actions and we report no error. If we attempted to match them in the reverse order, we would find matches for the redundant target accesses but fail to match the non-redundant ones, leading to an incorrect matching.

Once the analysis is complete, we analyze the unmatched actions as follows.

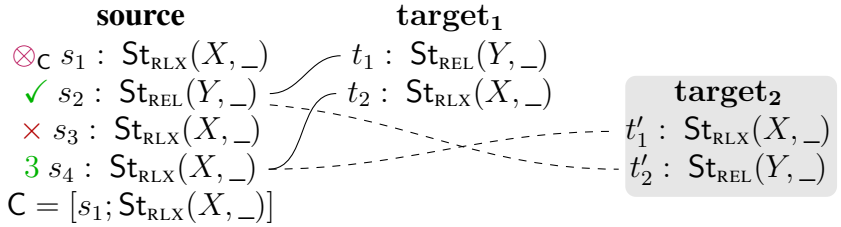
Analyze Introduced Actions An unmatched action in the target is an introduced action. We have three cases:

Writes/Updates: Introducing atomic writes or updates is generally unsound because a parallel thread may observe the additional update. However, introducing an immediately deletable non-atomic write is safe because any program that could observe the difference is anyway racy.

Reads: As bug #22514 shows, an introduced read is incorrect in C11, but allowed in our inferred LLVM model.

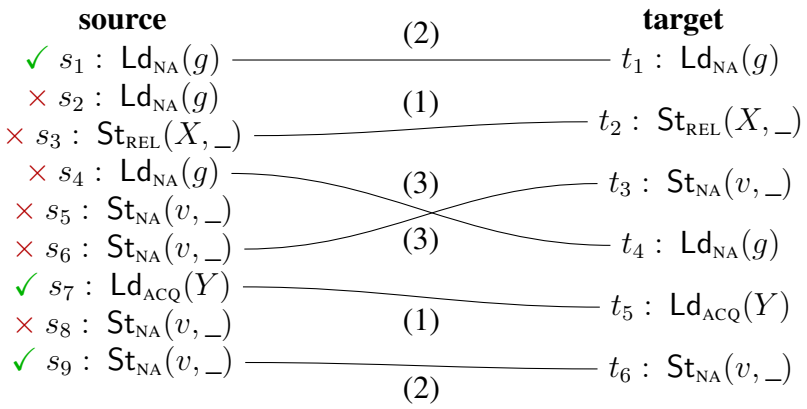
Fences: This is safe as it just adds synchronization.

Analyze Deleted Actions Unmatched actions in the source signify actions that have been deleted. For immediately deletable actions, there is nothing to check. For conditionally deletable ones, we check that the deletion preconditions are met. As the following example shows, checking the deletion preconditions is sometimes a bit subtle.



In this example, s_1 is conditionally deletable if C is satisfied. The action s_1 can be deleted only if s_2 and s_3 were reordered, but instead s_3 has been deleted! We therefore have to consider whether s_3 could have been reordered with s_2 before being deleted. In $\mathbf{target_1}$ the answer is *no* because after the reordering, s_3 is no longer deletable. Given that s_3 is deleted, we instead have to check that s_2 has been reordered with s_3's justifier (namely, s_4). So, the elimination of s_1 is *correct* in $\mathbf{target_2}$ but not in $\mathbf{target_1}$.

Example Finally we demonstrate the matching and analysis procedures on the following example:



First, we mark the source and the target actions as discussed in § 8.2.1 and then we match the synchronization source actions to the respective target actions. Thus we match s_3 with t_2 and s_7 with t_5. We proceed to the second step with the remaining non-deletable source actions. For s_1, the window is the singleton set $\{t_1\}$; so we match it with t_1. For s_9, the window likewise is the singleton set $\{t_6\}$; so we match it with t_6.

Finally, we try to match the remaining unmatched target actions, t_3 and t_4. To compute the search window, we identify the immediate predecessor *release* and immediate successor *acquire* of t_3 and t_4 which are t_2 and t_5 respectively. Thus, we match t_3 with s_6 and t_4 with s_4.

After the matching, we analyze the unmatched actions s_2, s_5, s_8. Since all three actions are *immediately deletable*, we conclude that deleting them is valid and hence the transformation is correct.

Dealing with Control Flow

We have so far discussed access matching for straightline code. In case of programs with control flow, there is more work to do. We proceed in two steps.

First, for each (loop-free) path in the target we identify the corresponding set of paths in the source. As we will explain, this is nontrivial because transformations may restructure the control flow, making it difficult to identify the corresponding source path for a given target path.

Second, for each pair of source and target paths, we identify the sequence of shared memory accesses and apply the matching discussed in § 8.2.1.

The validation is "Correct" if every target path is correctly matched by *all* corresponding source paths. Otherwise, the validator reports "Possible Error." We first provide a simple control flow graph (CFG) matching example and then discuss the general approach.

Example Figure 8.5 presents the CFGs corresponding to transformation of Figure 8.3 that witnesses the GVN bug. To make the two CFGs have matching entry and exit blocks, we

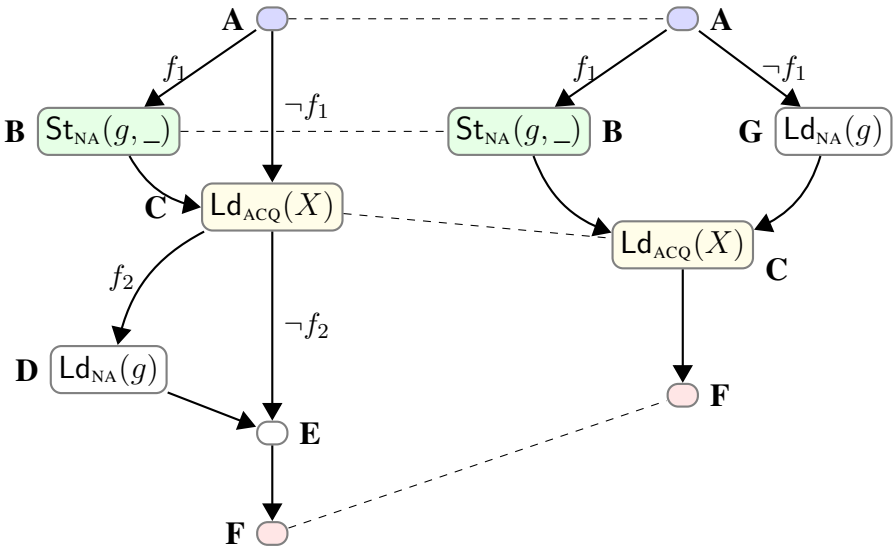


Figure 8.5: Transformation of the program in Figure 8.3.

append to both CFGs the empty **F** block. We represent the branch conditions in **A** and **C** by f_1 and f_2 respectively.

The blocks **D** and **E** are deleted and block **G** is introduced. The **A**, **B** **C**, and **F** basic blocks are matched because their name and path conditions match.

The target paths and the corresponding source paths are:

$$\mathbf{ABCF} \rightarrow \{\mathbf{ABCDEF}, \mathbf{ABCEF}\} \text{ and}$$
$$\mathbf{AGCF} \rightarrow \{\mathbf{ACDEF}, \mathbf{ACEF}\}.$$

Among the corresponding path pairs, matching the accesses of **AGCF** and **ACDEF** yields an error because of the unsafe reordering $\mathsf{Ld}_{\mathrm{ACQ}}(X); \mathsf{Ld}_{\mathrm{NA}}(g) \not\rightsquigarrow \mathsf{Ld}_{\mathrm{NA}}(g); \mathsf{Ld}_{\mathrm{ACQ}}(X)$.

Now we discuss the control flow matching technique.

Restructured CFG Matching Let $\{B_1 \ldots B_n\}$ be the set of basic blocks where B_1 and B_n are the entry and exit blocks respectively in the CFG. Also $\{f_1 \ldots f_{n-1}\}$ be the set of respective branch conditions of blocks B_1 to B_{n-1}. Given a path $P = B_1; \ldots B_j; B$ the path condition of P is denoted by $[\![P]\!] = f_1 \wedge \ldots \wedge f_j$.

Now consider that $\{P_1, \ldots, P_k\}$ be the set of paths from B_1 to B. We say that the reachability condition of B is $\Upsilon(B) = [\![P_1]\!] \vee \ldots \vee [\![P_k]\!]$. For example, $\Upsilon(\mathbf{E}) = (f_1 \wedge f2) \vee (f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f2) \vee (\neg f_1 \wedge \neg f_2)$ in Figure 8.5. A basic block B is not reachable if $\Upsilon(B) = false$.

We observe that even if an optimization restructures the CFG, the basic block names and the reachability conditions across the transformation is preserved by LLVM. Based on this observation the matching algorithm works as follows.

1. Match the basic blocks of the source and target CFGs by name and the respective reachability conditions.

2. Enumerate the set of paths from the function's entry node to the exit node in the target CFG.

3. For each such path, we find the set of corresponding source paths. Formally, we say that two paths P_S and P_T are corresponding if and only if their projections to the matched basic blocks are equal, $P_S|_{\mathsf{matched}} = P_T|_{\mathsf{matched}}$ as well as $[\![P_S]\!] \rightarrow [\![P_T]\!]$.

Once a path pair is found, we proceed to the access sequence matching algorithm of § 8.2.1.

Path matching in the presence of loops is well known to be difficult [69]. We handle a loop heuristically. We unroll the loop body once and then cut-off the loop backedge. Effectively this is similar to considering that the loop has at most two iterations. This suffices to preserve the cross iteration reachability among the scalar accesses.

8.2.2 LLVM-specific Matching Using Metadata (MD)

Our second approach uses a specific feature of the LLVM IR. The LLVM IR allows one to attach arbitrary "metadata" information along with the program constructs to preserve debugging information throughout compilation without affecting the optimization. We use instruction metadata to keep track of how the actions are moved by a transformation.

In brief, we instrument each action in the source program with a uniquely named metadata node. Next, we run the optimization pass(es) on the instrumented source program. The attached metadata nodes do not affect the optimizations but are preserved by LLVM's code motion transformations. Afterwards we use the metadata nodes to match the memory accesses and check that the transformation is correct.

In more detail, we have mildly modified LLVM to attach a unique `MDNode` metadata node at each shared memory access and fence at the beginning of the *opt* phase before any transformation takes place. Optimization passes are, in principle, allowed to drop any metadata nodes attached to instructions and even to arbitrarily change them. In practice, however, all the *opt* transformations neither depend on nor alter any metadata that they do not recognize. As a result, the transformations tend to move instructions along with their attached metadata. A couple of transformations drop unrecognized metadata and we have modified them so as to preserve it. The modified LLVM also merges the metadata when multiple identical instructions are combined (e.g., in CFG simplification). Finally, when a new instruction replaces an old instruction (e.g., in GVN), the metadata node is recreated from the old instruction.

Analysis of Matching

To illustrate the identification of errors after a metadata-based matching has been found, consider the dubious transformation in Figure 8.6. This transformation is incorrect for two reasons: (*i*) it violates the "roach motel" principle by reordering **c** and **d/f**; and (*ii*) it introduces a $\mathsf{St}_{\text{RLX}}(Z, \_)$ on a path where it previously did not appear. We will now explain how to catch these two errors in turn.

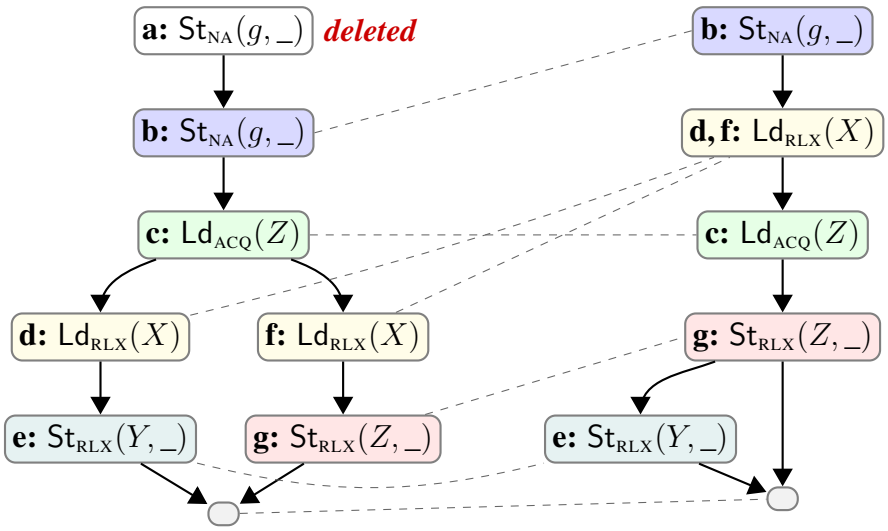


Figure 8.6: An unsafe transformation with matched accesses.

Reordering Correctness To catch the incorrect reordering of **c** and **d/f** in Figure 8.6, for any two non-commuting matched accesses a and b (i.e., when $a; b \not\rightsquigarrow b; a$), we have to check that if a precedes b in the source CFG, then a still precedes b in the target CFG. So, for a given CFG G, we define the set $\mathsf{OrderedPairs}(G)$ of all (a, b) such that a and b are both matched actions, there is a path from a to b in G, and $a; b \not\rightsquigarrow b; a$. We then check that $\mathsf{OrderedPairs}(\mathsf{CFG}_{\mathsf{src}}) \subseteq \mathsf{OrderedPairs}(\mathsf{CFG}_{\mathsf{tgt}})$.

Returning to the graphs in Figure 8.6, this check fails for nodes **c** and **d/f** indicating that the reordering is unsafe.

Matched Access Movement Correctness The second error in the transformation in Figure 8.6 cannot be caught with the previous analysis. The movement of the **g** access is incorrect not because it violates any reordering constraints but rather because it introduces a write along the **c** $\rightarrow$ **e** path. To catch these errors we compare the path conditions of the source and the target. If the source and target path conditions are different and the access is observable in another thread, then we report "Possible Error" and "Correct" otherwise.

Returning to our example in Figure 8.6, we deduce that the movement of **g** is incorrect because the entry block is not similar to the one on the left branch of the conditional. The merging of the **d** and **f** accesses is, however, correct in this sense because $\Upsilon(P_d) \vee \Upsilon(P_f) = \Upsilon(P_{d,f})$.

Introduced Actions If the target program has any observable unmatched write or update actions or, in the case of C11, also any unmatched reads, we report "Possible Error" considering such accesses as (incorrectly) introduced.

Deleted Actions If the source program has any unmatched action, we have to check that their deletion is justified.

Test Class	Model	End-to-End Validation			
		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD
(a) Straightline	LLVM	95	95	0	0
	C11	0	0	0	0
(b) With Branches	LLVM	64	74	0	3
	C11	13	39	1	27
(c) With Dead Paths	LLVM	58	74	0	2
	C11	6	40	0	25
(d) With Loops	LLVM	49	56	0	0
	C11	6	18	0	7
(e) Smaller Tests	LLVM	32	38	0	6
	C11	7	18	5	21

Table 8.1: Validation results of 100 tests and 11300 steps per class. Erroneous passes: † GVN and ‡ SimplifyCFG.

If the path condition is *false* then the action is not reachable and the deletion is justified. Otherwise, if the action is eliminable along every path from the entry node to the action then the deletion is correct and otherwise "Possible Error".

8.3 Evaluation and Discussion

We have implemented the two matching algorithms described in §8.2 and have applied them to validate transformations performed by LLVM. The results of our experiments are reported in Tables 8.1 and 8.2.

8.3.1 Experimental Setup

Test Case Generation To evaluate our validator, we developed a randomized test case generator that constructs programs with a desired number of accesses, approximate proportions of each access type, and so on. Each generated program consists of a single function containing multiple accesses of global atomic and non-atomic scalar variables intertwined with some local computations and random control flow determined by boolean variables. Initially, we synthesized four classes of tests:

(a) straightline programs,
(b) programs with dead-path-free conditional control flow,
(c) programs with conditionals including dead paths, and
(d) programs containing conditionals and *do-while* loops.

For each class, we generated 100 programs with 100 shared memory accesses each (roughly 85% non-atomic and 15% atomic) and, where applicable, 10 branch conditions. We restricted the shared memory accesses to only five global variables so that the compiler has plenty optimization opportunities. In fact, in all the generated programs, LLVM successfully performed

Test Class	Model	Stepwise Validation					
		llvm 3.6			llvm 3.7rc2		
		Non-id	CIM	MD	Non-id	CIM	MD
(a) Straightline	LLVM	927	95 †	95 †	835	0	0
	C11		0	0		0	0
(b) With Branches	LLVM	1209	64 †	74 †‡	1202	0	0
	C11		15 †‡	42 †‡		1 ‡	29 †‡
(c) With Dead Paths	LLVM	1442	57 †	73 †‡	1380	0	0
	C11		11 †	43 †‡		0	23 †
(d) With Loops	LLVM	1763	49 †	56 †	2152	0	0
	C11		7 †	20 †		0	10 †
(e) Smaller Tests	LLVM	779	32 †	38 †‡	782	0	0
	C11		7 †‡	24 †‡		6 ‡	21 †‡

Table 8.2: Validation results of 100 tests and 11300 steps per class. Erroneous passes: † GVN and ‡ SimplifyCFG.

some optimization to them. To ensure that there are no dead paths in case (b), we generate a different flag as the guard for each conditional. In cases (c) and (d), with high probability we reuse the same flag for multiple conditionals so that the compiler may recognize and eliminate some dead paths.

For test cases (a)–(d), we used a large number of memory accesses to test the efficacy of our validator and ensure that optimizations took place. Nevertheless, these large examples are not ideal for reporting errors, since in the end-to-end validation, errors in one optimization pass may be masked by a following pass. We therefore also generated some tests with a smaller number of accesses and control paths which reveal the actual bugs. We demonstrate one such set of 100 test cases (e), which revealed the reported bugs #22514 and #22708. Bug #22306 was identified by manual inspection.

Since our validator does not currently handle pointer and array accesses nor loop optimizations such as loop unrolling, we avoided generating programs with such accesses, and generated *do-while* loops, on which the undesired optimizations are not applicable. In principle, the validator could be extended to handle pointer and array accesses and identify such bugs.

Validation Parameters Our validation is parametrized by (*i*) the *LLVM version* tested, (*ii*) the *memory model* (either C11 or the LLVM model), (*iii*) the *validation approaches* (either compiler-independent matching (CIM), or metadata-based (MD)), and (*iv*) the *validation mode* (either end-to-end validation or stepwise validation for the individual transformations). As for the LLVM versions tested, we have chosen LLVM versions 3.6, which was the stable version when the work was done, and version 3.7rc2, a more recent version, in which our reported miscompilation bugs were fixed.

For each test program, we collect the unoptimized IR generated by the `clang++` frontend and optimize it with `opt -O3`. To perform stepwise validation, we used a LLVM command-line parameter that prints the IR before and after every optimization pass. It turns out, how-

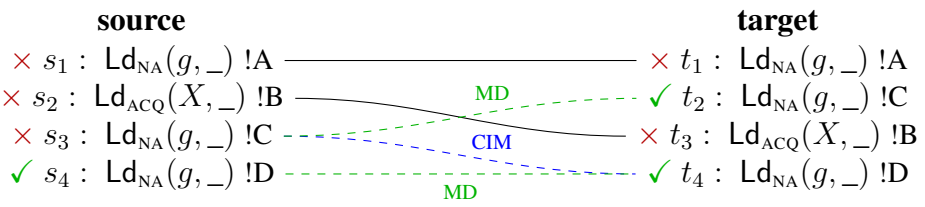
ever, that the IR after one optimization pass was not always identical to the IR before the next optimization pass. This is because in between LLVM performs various locally scoped optimizations (e.g., within a single basic block), and outputs only the affected IR, which is often difficult to relate to the whole function IR. We therefore ignored these partial IRs; we collected only IRs of the entire function CFG, and apply the validator to both the $\mathsf{IR}^{i}_{\text{pre}} \rightsquigarrow \mathsf{IR}^{i}_{\text{post}}$ and the $\mathsf{IR}^{i}_{\text{post}} \rightsquigarrow \mathsf{IR}^{i+1}_{\text{pre}}$ transformations. We do not validate the IR versions which are same and only validate the non-identical (Non-id) IR pairs. In total, there were 113 such validation pairs per test, only less than 3% of which actually changed the IR.

8.3.2 Observations

Tables 8.1 and 8.2 reports the results of our experiments. We make the following observations.

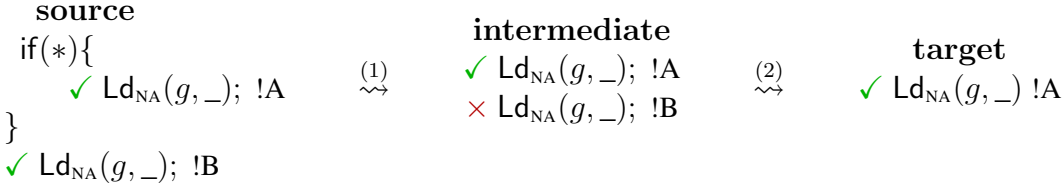
First, our CIM validator and the stepwise MD validator are extremely accurate: they report no errors in LLVM 3.7rc2 with respect to the LLVM model, but find plenty of errors in LLVM 3.6, and also some errors in LLVM 3.7rc2 with respect to the C11 model. We note that although many errors have been found, they are often caused by the same compiler bug. For example, all 95 errors found in straightline programs are due to bug #22708.

Second, the metadata-based validator (MD) finds more errors than CIM because it is less prone to having the effect of an invalid optimization being masked by its context. For example, consider the unsafe reordering $\mathsf{Ld}_{\text{ACQ}}(X); \mathsf{Ld}_{\text{NA}}(g) \not\rightsquigarrow \mathsf{Ld}_{\text{NA}}(g); \mathsf{Ld}_{\text{ACQ}}(X)$ applied to the following program:



CIM matches s_3 with t_4 and considers s_4 is deleted and t_2 is introduced. Since both the deletion and introduction are safe according to the LLVM model, CIM reports no error. MD instead matches s_3 with t_2 and reports an error. For the same reason, smaller tests are sometimes better at exposing errors with the CIM approach.

Third, CIM finds fewer errors end-to-end than with stepwise checking. This is because the effect of an erroneous transformation can be masked by a following transformation. The same, however, does not always hold for MD. There are several cases, where MD validates all individual steps, but reports and an end-to-end validation error. The following example illustrates those cases:



In the source the accesses are marked with unique metadata nodes which propagate along with the transformations (1) and (2). Both of the $\mathsf{Ld}_{\mathrm{NA}}(g)$ actions are non-deletable. In the LLVM model the action movement is allowed and thus (1) is correct. Also (2) is correct since $\mathsf{Ld}_{\mathrm{NA}}(g,\_)$!B deletion is safe. However, although both (1) and (2) are safe, the end-to-end $\mathsf{Ld}_{\mathrm{NA}}(g,\_)$!B deletion is reported as a "Possible Error" since MD finds that the non-deletable $\mathsf{Ld}_{\mathrm{NA}}(g,\_)$!B is deleted.

Thus, although MD can be more precise, some of the errors it reports especially in the end-to-end mode are false positives. False positives arise because LLVM occasionally drops or mixes up metadata information, e.g. when creating or merging ϕ nodes, or in cases such as discussed previously. We therefore consider our two validation approaches complimentary: MD is better for validating individual optimizations, whereas CIM is better for end-to-end validation.

In our experiments, we observed that LLVM does not normally perform eliminations and reorderings of atomic accesses. It marks the atomic accesses as *'volatile'* in the IR to avoid any deletion or reordering among them. While this strategy facilitates achieving correctness, various optimization opportunities are lost, which is considered as a potential place for improvement (see LLVM documentation [43]). In contrast, non-atomic shared accesses are heavily optimized (e.g., reordered with atomic accesses and/or deleted).

Out of the roughly 40 LLVM passes applied (some multiple times each), we observed that only 13 actually affected the test programs: SROA, early CSE, redundant instruction combination, function integration/inlining, expression reassociation, dead store elimination, GVN, CFG simplification, jump threading, SCCP, value propagation, LCSSA, and natural loop canonicalization. In two of these passes, we have found errors: GVN and CFG simplification.

Finally, there are no validation errors according to the C11 model for straightline programs, but several for programs with control flow. This can be explained by observing that these errors arise because of the introduction of speculative memory loads. LLVM, of course, does not introduce loads needlessly: these get introduced as a result of restructuring the program's CFG.

Chapter Summary In this chapter we discussed the translation validator we developed to validate LLVM optimizations with respect to concurrency. Using the validator we identified certain concurrency bug in LLVM which were reported and were fixed. In the next chapter we discuss the related work; we discuss various results relevent in relaxed memory concurrency and correctness in concurrency compilation.

9 Related Work

In this chapter we discuss the research efforts in defining relaxed memory concurrency semantics especially to address 'out-of-thin-air' behavior. Next, we discuss various approaches of compiler correctness.

9.1 Semantics for Handling 'Out-of-Thin-Air'

As explained in the introduction, defining an adequate concurrency model that does not allow *out-of-thin-air* behaviors has been a longstanding research challenge, and has led to a number of proposals. In addition to our proposed WEAKEST and WEAKESTMO models, we are aware of the following models that correctly differentiate CYC, LB, and LBfd:

- The operational Java memory model proposed by Manson et al. [46].
- AE-justification by Jeffrey and Riely [31].
- 'Bubbly' semantics by Pichon-Pharabod and Sewell [55].
- Promising semantics by Kang et al. [33].

We have already discussed the Java memory model in § 2.4.4. The Java memory model also provided 20 causality benchmarks, which became a standard benchmark for comparing memory models. We have already discussed these benchmarks in §6.1. We now elaborate on the models proposed by Jeffrey and Riely [31] and Pichon-Pharabod and Sewell [55] along with their respective limitations. Next, we discuss the limitations of the promising semantics [33], as we have already discussed the scheme in detail in Chapter 4.

9.1.1 AE-Justification

Jeffrey and Riely [31] defined a model for a fragment of Java based on event structures. In this model an event structure captures multiple *configurations* where a configuration is conflict-free and downward closed set of events in the event structure. A configuration is *justified* when every read event in the configuration is justified by some write in the same justification. Note that the configuration is similar to the execution in the Java memory model.

In order to get a valid configuration the scheme introduce a two player games. In this game a configuration is selected from an event structure and is extended by adding one event at a time from the event structure. Consider that the game start with a configuration C. The player's goal is to extend configuration C to configuration D. The opponent extends configuration C to configuration C' where the new read events are acyclically justified. The player then

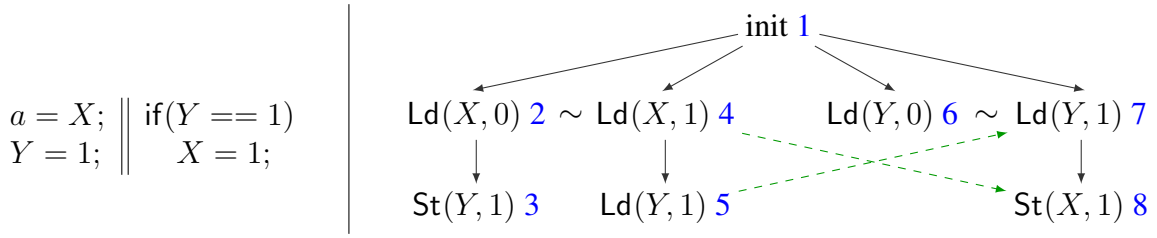
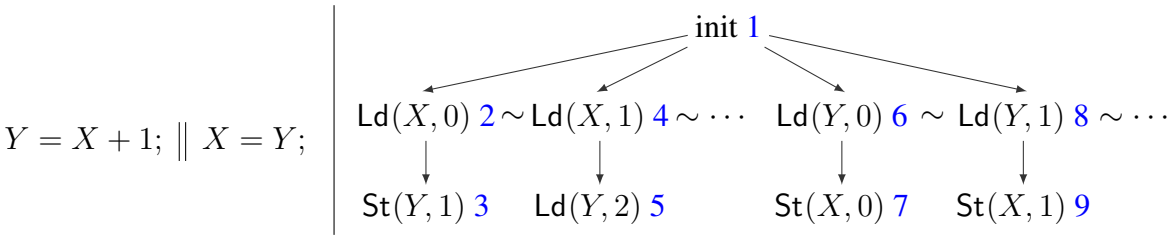


Figure 9.1: Jeffrey and Riely [31] model allows $a = b = 1$ outcome in LB program.

extends C' to C'' where the new reads are also acyclically justified. Continuing with the alternative turns, the player wins if C'' justifies D, and otherwise the opponent wins. If the player has a winning strategy for this game then the initial configuration C Always Eventually (AE) justifies final configuration D. Thus in a justified final configuration each newly added read event has to be AE-justified: for all moves of the opponent (choosing how to continue executing the program), there must exist an extension containing a write whence the read gets its value.

Now we explain the $a = b = 1$ outcome on the LB program where the locations are initialized to zero in Figure 9.1. The game starts with $\emptyset$ configuration. The player extends $\emptyset$ configuration to $\{1, 7, 8\}$ and $\emptyset$ configuration AE-justifies $\{1, 7, 8\}$ configuration. The opponent can extend $\emptyset$ configuration to $\{1, 2, 3, 6\}$ or $\{1, 2, 3, 7, 8\}$. Both of these configurations include 3 (i.e. St$(Y, 1)$) which justifies 7 (i.e. Ld$(Y, 1)$). As a result, the player do not add any event to justify $\{1, 7, 8\}$. The $\{1, 7, 8\}$ can be extended to $\{1, 4, 5, 7, 8\}$ and $\{1, 7, 8\}$ configuration AE-justifies $\{1, 4, 5, 7, 8\}$. Note that whenever opponent extend to a configuration which is acyclically justified from $\{1, 7, 8\}$, such configuration contain event 8. In this case event 8 (i.e. St$(X, 1)$) justifies event 4 (i.e. Ld$(X, 1)$). As a result, the player does not have to add any event to justify $\{1, 4, 5, 7, 8\}$ and we get the desired configuration to justify the $a = b = 1$ outcome in the LB program.

Now we consider another example where X and Y are initialized to zero:



In this program no execution can have $Y = 2$. Considering the Jeffrey and Riely [31] model, the player cannot extend $\emptyset$ configuration to $\{1, 4, 5\}$, as the opponent in this case extend $\emptyset$ configuration to $\{1, 2, 3, 6, 7\}$. The configuration $\{1, 2, 3, 6, 7\}$ has no St$(X, 1)$ event which can justify event 4 (i.e. Ld$(X, 1)$). As a result, $Y = 2$ is not allowed in this program. Note that the above discussed example is a reduced version of RNG program from § 3.2.1 and in consequence Jeffrey and Riely [31] model assigns the correct semantics to the RNG example.

Thus Jeffrey and Riely [31] model justifies $a = b = 1$ outcome in LB program as discussed above and disallows $a = b = 1$ in CYC program. In addition the model justifies the outcome of a number of Java litmus tests [45] with certain exceptions which we discuss shortly.

There are some results established on this model. The model ensures data race-freedom guarantees; "*if all sequentially consistent configurations are data-race-free, then all well-justified configurations are sequentially consistent*". The model also support program with release acquire fences and the data-race-freedom guarantee also extends to the well-fenced programs; " *if all sequentially consistent configurations are data-race-free, then all well-fenced configurations are sequentially consistent.* ". Moreover, the model facilitates invariant reasoning on event structure. The results are discussed in detail by Jeffrey and Riely [31].

Although Jeffrey and Riely [31] address a number of issues of relaxed memory concurrency, the model has some limitations, the first of which is quite severe.

Read-Read Reordering is Unsound The model fails to validate the reordering of independent read events, and therefore cannot be compiled to Power and ARM without additional fences. Consider the Java test case 7 [45]:

$$\begin{array}{c} \text{Initially } X = Y = Z = 0 \\ \begin{array}{l} r_1 = Z; \\ r_2 = X; \\ Y = r_2; \end{array} \left\| \begin{array}{l} r_3 = Y; \\ Z = r_3; \\ X = 1; \end{array} \right. \end{array} \quad \rightsquigarrow \quad \begin{array}{c} \text{Initially } X = Y = Z = 0 \\ \begin{array}{l} r_2 = X; \\ Y = r_2; \\ r_1 = Z; \end{array} \left\| \begin{array}{l} r_3 = Y; \\ Z = r_3; \\ X = 1; \end{array} \right. \end{array}$$

In the first thread the reads of Z and X are reordered. Following the Jeffrey and Riely [31] model, the outcome $r_1 = r_2 = r_3 = 1$ is forbidden in the source program but is allowed in the target program. The detailed explanation is in Jeffrey and Riely [31, §7].

On the contrary, our model allows the $r_1 = r_2 = r_3 = 1$ outcome in the source program as shown in Figure 6.1 similar to that of the target program and hence reorderings of these reads is a sound transformation in our model.

Jeffrey and Riley, however, sketch an extension of their model that seems to fix this problem. It remains unclear whether the extension satisfies DRF-SC and can be compiled to hardware with the intended mappings.

Weaker Coherence Following the intended Java semantics, the Jeffrey and Riely [31] model does not enforce coherence on non-atomic accesses. As a result, the model allows the non-coherent outcome $a = 2 \wedge b = 1$ in the following litmus program taken from [45]:

$$\begin{array}{l} a = X; \\ X = 1; \end{array} \left\| \begin{array}{l} b = X; \\ X = 2; \end{array} \right. \qquad \text{(RW2Loc)}$$

According to Jeffrey and Riely [31], the model requires to detatch the *causal* order from visibility to enforce coherence. However, it is unclear how to do so and hence the model cannot be directly applied to C11 as of now.

Computational Cost AE justification is computationally very expensive in comparison to our more basic existential justification. Enumerating all possible executions of a program is computationally infeasible, because one has to prove an assertion with N quantifier alternations where N is the length of the execution.

9.1.2 "Bubbly Semantics"

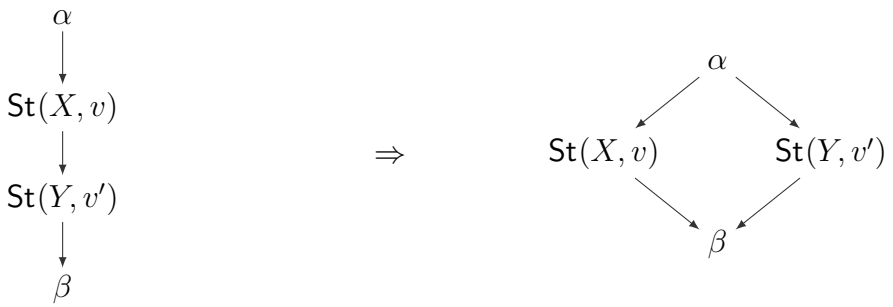
Pichon-Pharabod and Sewell [55] proposed a model for a fragment of C11 based on plain event structures with only program order and conflict edges. They define an operational semantics over such event structures that iteratively constructs an execution. It does so by either committing a minimal event from the structure or by performing one of set of pre-determined compiler optimizations that rewrites the remaining event structure. For instance, *deordering* and *merging* are such two steps which can be performed on an event structure. Deordering captures the effect of reordering independent memory accesses and merging captures the effect of redundant access elimination transformation. We elaborate these steps with example.

Deordering Deordering eliminates the program order which results from syntactic order but no semantic dependency. Consider the first thread (i.e. $\{a = X; Y = 1;\}$) of the LB program:

$$\begin{array}{l|l} a = X; \\ Y = 1; \end{array} \left\| \begin{array}{l} b = Y; \\ \mathsf{if}(b) \\ \quad Y = 1; \end{array} \right. \qquad \begin{array}{ccccccc} & \mathsf{Ld}(X,0) & \sim & \mathsf{Ld}(X,1) & \sim \cdots & \mathsf{Ld}(X,v) & \sim \\ & \downarrow & & \downarrow & & \downarrow & \\ & \mathsf{St}(Y,1) & & \mathsf{St}(Y,1) & & \mathsf{St}(Y,1) & \\ & & & \Downarrow & & & \\ \mathsf{St}(Y,1) & \mathsf{Ld}(X,0) & \sim & \mathsf{Ld}(X,1) & \sim \cdots & \mathsf{Ld}(X,v) & \sim \end{array}$$

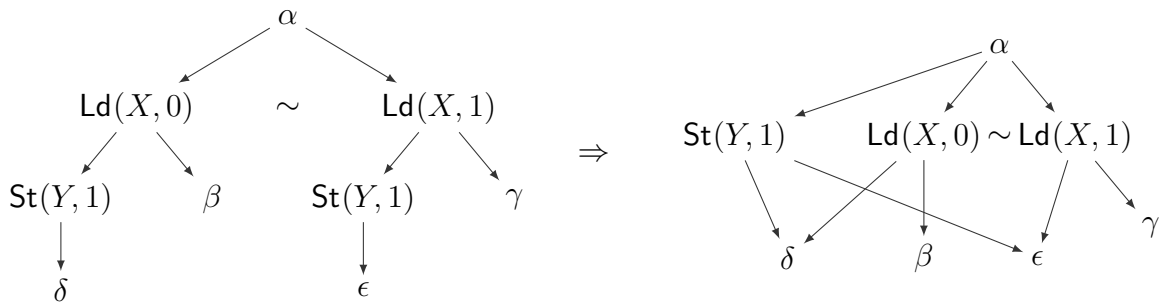
In this program, there are no dependencies between the accesses in the first thread. As a result, we can transform the event structure where we deorder $\mathsf{St}(Y,1)$ with the conflicting reads of X. More generally, deordering can be performed in a number of scenarios, which we explain by example.

The simplest case is to deorder a write event with another write event as in the following example.

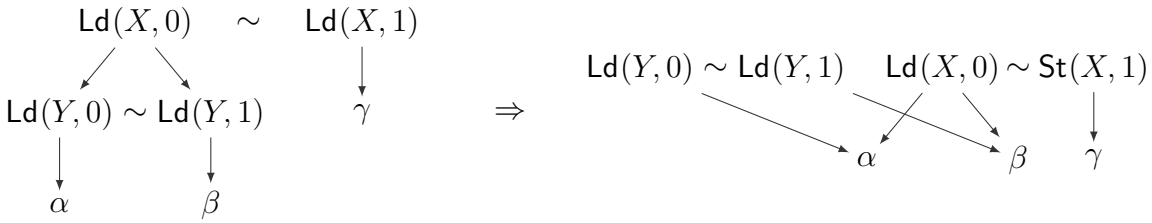


As shown, the write of Y is deordered with the write of X. In the source event structure, there is a program order from $\mathsf{St}(X, v)$ to $\mathsf{St}(Y, v')$. After the reordering, the program order from $\mathsf{St}(X, v)$ to $\mathsf{St}(Y, v')$ is eliminated.

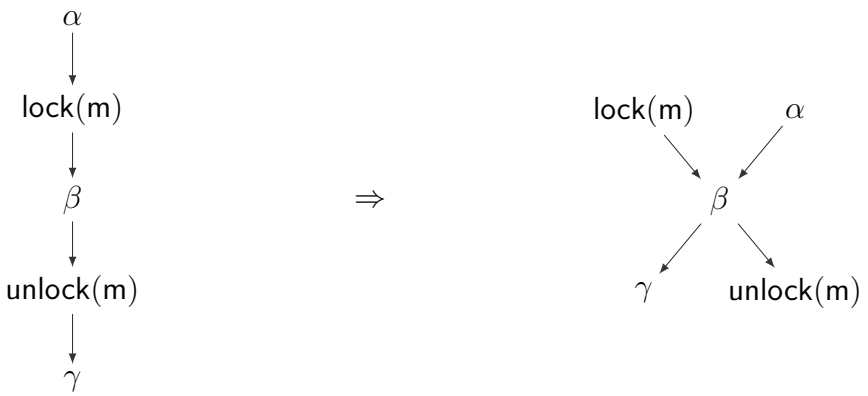
A more complicated case is the deordering of a set of write events with a set of earlier read events. In the following example, the $\mathsf{St}(Y, 1)$ events are deordered with the set of reads on X. The target event structure is generated as follows.



Further, a set of read events can be deordered with another set of read events. For example, we reorder set of reads on Y with the set of reads of X as shown below.

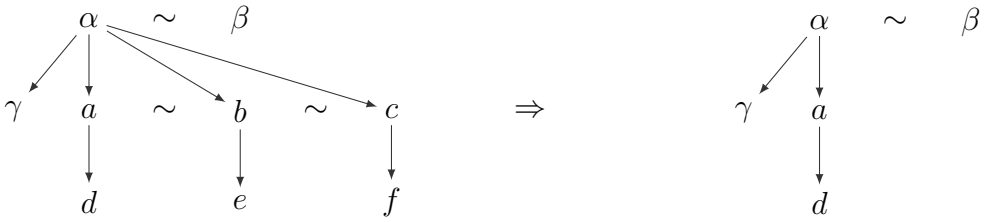


The model allows *roach-motel deordering* where (1) a 'lock' operation is deordered with respect to the program order predecessors or (2) program order successors of an 'unlock' operation are deordered with respect to the respective 'unlock' operation. Thus roach-motel deordering steps extend a critical section.

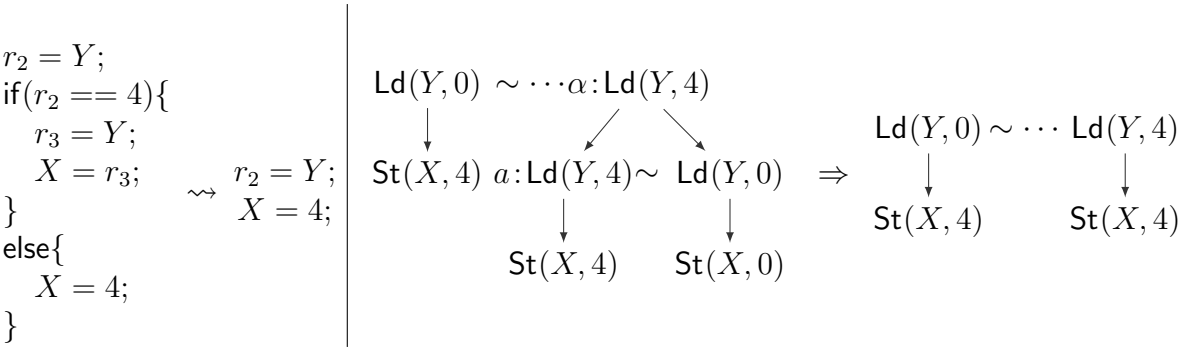


Merging In a merging step multiple events are merged to an event. A merging can be a forward or a backward merging.

Forward Merging A forward merging step merges an event e_2 to e_1 where e_2 is program order successor of e_1. Now consider event e_3 in conflict with e_2 and e_3 is program order successor of e_1. In this case the step also removes e_3 and its program order successors as these alternative events cannot be executed anymore. Thus the event structure transformation is as follows.



For instance, the forward merging steps captures the redundant read elimination transformation as follows.



Given the source event structure, the forward merging step transforms it to the target event structure by (1) merging the event $a : \mathsf{Ld}(Y, 4)$ to $\alpha : \mathsf{Ld}(Y, 4)$ and (2) discarding the $\mathsf{Ld}(Y, 0)$ and $\mathsf{St}(X, 0)$ which are in conflict with $a : \mathsf{Ld}(Y, 4)$.

Backward Merging In this step an event is merged to its multiple successors. This transformation step is applied to eliminate overwritten write event in an event structure.

In the source event structure the write $\mathsf{St}(X, 1)$ is overwritten by its program order successors events: $\mathsf{St}(X, 2)$ and $\mathsf{St}(X, 3)$. Hence the backward merging step merges $\mathsf{St}(X, 1)$ to $\mathsf{St}(X, 2)$ and $\mathsf{St}(X, 3)$ events to generate the target event structure.

More Transformations In addition to deordering and merging steps, Pichon-Pharabod and Sewell [55] propose some more transformations on event structure such as execution step, value range speculation step, and so on. An execution step selects a sequence of events ordered by program order relation and discard its conflicting branches. A value range propagation step marks *dead* events and discard them as they cannot take place in any program execution. The details of these steps are in Pichon-Pharabod and Sewell [55].

Results The model interleaves program execution and program transformations in an event structure to capture the relaxed behaviors.

We observe three main results in this approach. First, the model supports a number of reordering and elimination transformations in the C11 model. Second, the semantics also models the effect of undefined behavior for the racy programs. Third, the bubbly semantics evaluates the model on the Java testcases [45] in Pichon-Pharabod [56]; the model can handle the testcases except the ones with 'loop' and 'volatile' accesses.

Limitations The resulting model is quire complicated and, as such, does not have much metatheory developed about it. For example, the model does not provide any DRF guarantee. The model does not address the C11 release, acquire, or SC accesses. it also fails to explain some weak program behaviors (cf. the ARM-weak program of [33]). Moreover, the handling of Java testcases 19, 20 is ad-hoc. These testcases require thread sequentialization which is not supported by the model. They claim they can handle these cases by extending the model by introducing thread *join* operator. However, introducing thread sequencing would allow the forbidden behavior in test case 5 in their model.

9.1.3 Promising Semantics

Kang et al. [33] defined the promising semantics (PS) which we have discussed in Chapter 4. The model comes with a number of results which we are able to exploit for our WEAKEST model. It has also led to some follow up work, which proved correctness of compilation to ARM [58, 59] and the correctness of a program logic over it [70]. Nevertheless, as discussed, the promising semantics also has a number of shortcomings, which are difficult to resolve because of the model's complexity and brittleness. We believe there are three main reasons for this complexity:

1. Instead of execution graphs, PS uses timestamps and messages to represent executions. While this may be closer to a hardware implementation, the particular representation of timestamps is irrelevant for the semantics.

2. PS allows promising a write at any point during execution, and certificates of arbitrary length. As a result, it is rather difficult to use PS as an execution oracle.

3. Finally, in order to handle RMWs, PS quantifies over all 'future' memories in the certification of promises [33, §3]. This is not only inhibits using PS as an execution oracle, but also has quite odd consequences regarding the set of allowed behaviors.

Its shortcomings include (a) admitting some questionable behaviors (e.g., that of Coh-CYC); (b) not supporting sequentialization and other global optimizations; (c) providing overly strong semantics of RMWs, whose compilation to ARMv8 incurs an unintended performance cost (cf. FADD); and (d) not supporting SC accesses.

Our work provides solutions to problems (a), (c), and (d), albeit our solution for (c) is not supported by a compilation correctness proof. Similar to PS, we do not support global optimizations.

9.2 Compiler Correctness

Compiler correctness is a long-standing research topic and there are various approaches that aim to achieve correct compilation. In this discussion, we categorize them into verified compilation, translation validation, and compiler testing schemes.

9.2.1 Verified Compilation

In verified compilation, the compiler comes together with a mechanized proof ensuring that whatever transformations it performs are correct. The most prominent such compiler is CompCert [40]. While verified compilation ensures correct compilation by construction, it requires significant effort and expertise to perform the proofs. Because of this difficulty, the applicability of verified compilers so far been limited.

Sevcík et al. [67] and Beringer et al. [15] have extended the Compcert verified compiler to support concurrency. The CompcertTSO compiler extends Compcert's intermediate language clight to clightTSO. The intermediate language clightTSO introduces concurrency primitives along with TSO semantics to perform provably correct compilation to the x86-TSO architecture. Beringer et al. [15] have developed the concept of language-independent compiler correctness for shared-memory concurrency using *logical simulation relations* proof techniques based on Compcert compiler.

Considering the subtle complexity of C11 concurrency and the difficulty of developing a verified compiler, it is not surprising that there is no verified compiler for the C11 concurrency till date. The first step towards verified optimizing compilers is to identify the sound program transformations under a given weak memory model.

Sevcík [66] first studied this problem in the context of a simple DRF memory model by considering a set of abstract transformations. As a result, Sevcík [66] identified the safe reordering transformations among non-atomic or plain accesses. The model also identified safe reordering of non-atomic accesses with Java 'Volatile' accesses as well as with 'lock, 'unlock' operations. In addition, Sevcík [66] identified a set of safe eliminations of redundant non-atomic accesses.

Later, Morisset et al. [48] and Vafeiadis et al. [74] studied the same problem in the context of the C11 memory model. Morisset et al. [48] identified the safe reorderings of an access pair where one of them is a non-atomic and the other is a non-atomic or C11 atomic access. Vafeiadis et al. [74] identified the safe reordering and elimination transformations for both non-atomic as well as atomic accesses. As a result, Vafeiadis et al. [74] studied program

transformations in greater detail. Vafeiadis et al. [74] showed that many desired transformations are not allowed in the original C11 model. They evaluate the reordering transformations including atomic access pairs and identified the safe reorderings. Moreover, they evaluate the elimination transformations and identified safe eliminations including the cases where elimination of a C11 atomic access are correct.

They observe that original C11 model does not allow many desired transformations some of which we have already demonstrated in §2.4.5. Hence, Vafeiadis et al. [74] propose fixes in C11 model and evaluate the transformations on the proposed fixes. However, none of the fixes were successful to justify all the desirable transformations. These results demonstrated that 'per-execution' based models do not suffice to meet all the requirements of C11 concurrency and constitute the motivation of alternative formalizations including the proposed approaches in this thesis.

9.2.2 Translation Validation

Translation validation is a simpler verification approach that is typically decoupled from the original compiler development and is reusable for multiple compilers and languages. Given a run of the compiler, it just checks if the target program refines the source program. Since this is undecidable in general, one often relies on clever heuristics [57, 50, 71] or checks a simpler property that may or may not imply refinement. An alternative scheme is to instrument the compiler to augment program to facilitate the validation [49]. The MD validator follows this scheme and instruments LLVM. However, compared to Namjoshi and Zuck [49], the instrumentation effort and the extracted information is significantly less in our MD validation and is easily replicable across compilers.

Prior to this work discussed in Chapter 8, no validation work for C11 concurrency compilation existed. Our approach can be seen as translation validation with the crucial difference that we only check that the memory access sequences in the two programs correctly match up and not program equivalence or refinement. We catch the concurrency-related errors which are not identifiable by the existing sequential validators. However, presently our approach does not catch the translation errors on thread-local variables.

9.2.3 Compiler Testing

Another approach for improving compiler trustworthiness is extensive testing. Here, many automatically generated test programs are compiled with and without optimizations, executed, and their results are compared to check for optimization errors. Although testing does not ensure correctness, it has been extremely effective at finding bugs [76, 39].

Testing concurrent program compilation has been explored in different scenarios [76, 39, 41, 48]. Yang et al. [76] and Le et al. [39] have employed testing techniques and have reported hundreds of compilation errors in GCC and LLVM. Lidbury et al. [41] have also used testing approaches to check OpenCL compilers and have found over 50 bugs in commercial compilers. Although these bugs were exposed by compiling concurrent programs, manually reducing the test cases revealed that none of the bugs found were actually inherently concurrency-related.

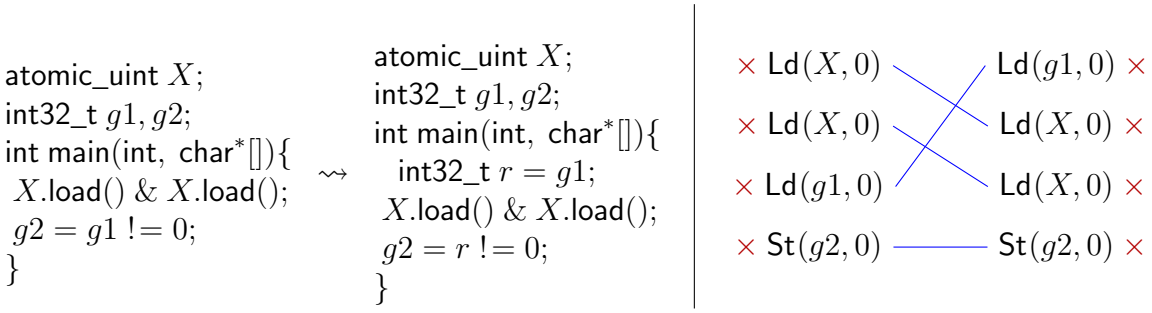


Figure 9.2: Unsafe reordering and matching of x86 traces in Morisset et al. [48].

Following the success of automated testing for identifying sequential compilation errors, Morisset et al. [48] have applied testing to check for C11 concurrency errors in GCC and have identified a number of concurrency bugs. Our approach is closely related to that of Morisset et al. [48] but has important differences. We thus elaborate the techniques in Morisset et al. [48] and then compare with our approach.

Morisset et al. [48] identifies the correct reordering and elimination transformations for non-atomic. Next, they instrument the compiled programs so as to record the sequence of memory accesses performed, and then try to match the sequence of accesses among the two versions of the program with and without optimization. To do so, they mark the accesses in the original and optimized access sequences to identify whether an access is *eliminable* or *non-eliminable*. Finally, they match the access and eliminate the unmatched eliminable accesses. The matching is successful if all accesses in both traces are either matched or eliminated. For example, consider the example from Morisset et al. [48] in Figure 9.2. In this C program a is an atomic and $g1$, $g2$ are non-atomic variables. Expression X.load() denotes an SC load operation of location X.

The GCC compiler [1] moves the read of $g1$ before the SC read of X which is not a safe transformation. The original and optimized programs are instrumented and executed which result in respective traces as shown above. The accesses in these traces according to Morisset et al. [48] are non-eliminable. The accesses are matched and the matching exposes the incorrect reordering of reads of X and $g1$. Based on the matching, Morisset et al. [48] identified a number of concurrency bugs in the GCC compiler which were reported and fixed.

We now discuss the Difference between Morisset et al. [48] and our Approach. The major one is that we compare two C11 program CFG structures, whereas Morisset et al. [48] compare two particular executions. Thus, our matching algorithms are sufficiently more complicated because they consider the programs' CFGs, which may be structurally dissimilar because of transformations.

A second difference is that we perform validation at the compiler IR level, whereas Morisset et al. [48] do it at the assembly level. Matching at the assembly code is problematic because the conversion to assembly loses a lot of the information present at the IR level, such as the memory order annotations. For example, consider the two transformations:

$$\mathsf{St}_{\text{RLX}}(X,\_);\mathsf{St}_{\text{RLX}}(Y,\_); \quad \rightsquigarrow \quad \mathsf{St}_{\text{RLX}}(Y,\_);\mathsf{St}_{\text{RLX}}(X,\_);$$
$$\mathsf{St}_{\text{REL}}(X,\_);\mathsf{St}_{\text{REL}}(Y,\_); \quad \rightsquigarrow \quad \mathsf{St}_{\text{REL}}(Y,\_);\mathsf{St}_{\text{REL}}(X,\_);$$

The first transformation is valid, whereas the second one is not. At the assembly level, however, the St_{REL} and St_{RLX} events are indistinguishable: they are both `MOV` instructions. Thus, by performing matching just at the assembly level, one will necessarily miss a number of bugs or will report many false positives. Matching at the IR level enables us to provide better precision and cover a broader set of transformations.

10 Conclusion

The results obtained in the thesis lead to a number of research directions. We discuss some of the possible future directions to come up with better semantics for relaxed memory concurrency, to use the proposed semantics in reasoning about programs, and to develop improved translation validator.

Our proposed formalizations have addressed a number of issues in relaxed memory concurrency. However, the proposed semantics can be improved in following aspects.

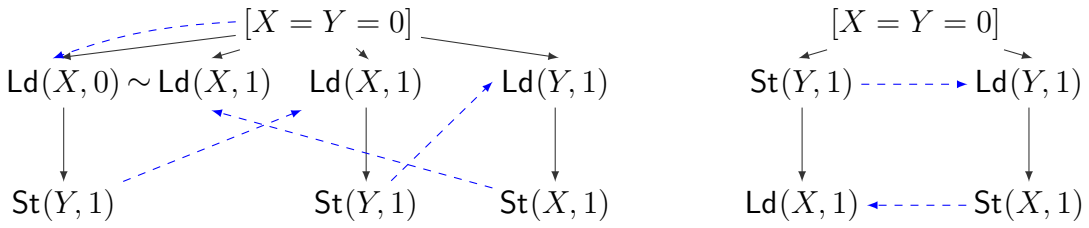
10.1 Proving Correctness of the Most Efficient Mappings to ARM & PowerPC

The mappings to PowerPC and ARM architectures proposed in this thesis leverage the mapping results of RC11 [37]. However, the proposed mappings by [37] for C11 to PowerPC and ARM architectures are sub-optimal. As shown in Figure 7.2, the mappings of C11 'relaxed' loads to PowerPC and ARMv7 introduces a fake conditional branch, that is $\mathsf{Ld}_{\text{RLX}} \rightsquigarrow \mathsf{ld}; \mathsf{cmp}; \mathsf{bc}$. Ideally, however relaxed loads should be mapped to plain loads without a fake branch: i.e., $\mathsf{Ld}_{\text{RLX}} \rightsquigarrow \mathsf{ld}$. We believe it should be possible to establish the efficient mappings from WEAKESTMO to these architectures. To achieve this, one could try to prove correctness of the intended compilation scheme to IMM [59], an intermediate memory model that has efficient compilation schemes to ARM and PowerPC and was used to prove the correctness of compilation from the promising semantics.

10.2 Sequentialization

As mentioned earlier sequentialization is unsound in our proposed models as well as in promising semantics [33]. Consider the following counterexample taken from [33, §6].

$$\begin{array}{l} \text{Initially } X = Y = 0; \\ \begin{array}{l|l|l} a = X; \quad \text{// 1} & & \\ \mathsf{if}(a \neq 1)\{ & Y = X; & X = Y; \\ \quad X = 1; & & \\ \} & & \end{array} \end{array} \overset{(1)}{\rightsquigarrow} \begin{array}{l} \text{Initially } X = Y = 0; \\ \begin{array}{l|l} a = X; \quad \text{// 1} & \\ \mathsf{if}(a \neq 1)\{ & \\ \quad X = 1; & X = Y; \\ \} & \\ Y = X; & \end{array} \end{array} \overset{(2)}{\rightsquigarrow}$$



(a) Source (b) Target

Figure 10.1: Sequentialization is unsound in WEAKEST and WEAKESTMO.

$$\begin{array}{l|l} \text{Initially } X = Y = 0; & \\ a = X; \quad // \, 1 & \\ \mathsf{if}(a \neq 1)\{ & \\ \quad X = 1; & X = Y; \\ \} & \\ Y = 1; & \end{array} \quad \overset{(3)}{\rightsquigarrow} \quad \begin{array}{l|l} \text{Initially } X = Y = 0; & \\ Y = 1; & \\ a = X; \quad // \, 1 & \\ \mathsf{if}(a \neq 1)\{ & X = Y; \\ \quad X = 1; & \\ \} & \end{array}$$

The behavior in question is whether $a = 1$ is possible in any execution. We observe that by a sequence of transformations as shown may result in this outcome. Transformation (1) sequentializes the second thread to the first thread. Transformation (2) performs a speculative value propagation and assigns 1 to location Y. As a result, the store of Y does not depend on X anymore. As a result, transformation (3) reorders $Y = 1$ before the accesses of X. Considering this target program there may an interleaving where Y is written 1 followed by a read of Y value 1 and write 1 to X in the second thread and then X reads 1 in the first thread which results in $a = 1$.

Kang et al. [33, §7] explains that promising semantics does not allow $a = 1$ in the first thread, but allows $a = 1$ in the target program. As a result, sequentialization is unsound in promising semantics. Similarly, as shown in Figure 10.1, our proposed models also prohibits $a = 1$ in the source event structure as $\mathsf{Ld}(X, 1)$ is not a *visible* event and allow $a = 1$ in the target event structure.

In the future, we want to develop formal models similar to WEAKEST and WEAKESTMO which would allow sequentialization.

10.3 Reasoning about Programs

Further ahead, for WEAKESTMO to be established as a good programming language consistency model, we would have to develop techniques to reason about concurrent programs running in the WEAKESTMO model. There are two types of reasoning techniques that are relevant: program logics and model checking.

Program Logics In terms of *program logics*, there are several works handling different fragments of RC11 (e.g., [73, 25, 26, 72, 32]). Their soundness proofs, however, rely heavily

on the acyclicity of $\mathsf{po} \cup \mathsf{rf}$ and it is unclear whether they can be adapted to the weaker setting of WEAKESTMO. Among these logics, RSL [73] should be sound under our model, because Svendsen et al. [70] proved the soundness of a variant of it over the promising semantics. For the more advanced logics, however, the counterexample of Doko and Vafeiadis [26] shows that certain adaptations will be needed.

Model Checking In terms of *model checking* for weak memory concurrency, again there are a number of works that assume $\mathsf{po} \cup \mathsf{rf}$ acyclicity (e.g., [34, 6]). There are also tools that do not require $\mathsf{po} \cup \mathsf{rf}$ acyclicity [52, 8, 5]. These tools, however, use more expensive state enumeration techniques, and are significantly slower than the tools that assume $\mathsf{po} \cup \mathsf{rf}$ acyclicity. It remains to be seen whether similar model checking approaches can work for models based on event structures, and if so, how much slower they would be in comparison to the existing tools.

10.4 Translation Validation

Concerning the validator, we developed a technique for validating LLVM optimizations with respect to concurrency. Our validator has proved useful in finding concurrency-related compiler bugs, and could in principle be integrated in the compiler's regression testing suite. Nevertheless, doing so in a useful fashion would require more implementation work. In addition, it would be nice to extend the validator in the following directions.

Handling Thread-Local Transformation Presently the validator focuses exclusively on matching the shared memory accesses with one another, and assumes that any other thread-local transformations are correct. A natural extension would be to also analyze the correctness of the thread-local transformations so that we can prove the overall correctness of a transformation of a given program.

Language Features The validator currently supports a fragment of C/C++ language features. For instance, we do not handle array accesses, pointers, and so on. It is because our goal has been to validate simpler programs on which compilers perform aggressive transformation. One can extend the validator with advanced language features so that we can validate programs with advanced language features.

Validating Loop Transformations The validator presently has very primitive support for handling programs with loops. To handle more advanced loop transformations (e.g., loop unrolling, loop switching), we require to incorporate advanced analysis techniques to be able to detect what transformations took place. Alternatively, we could perhaps modify the compiler to output what loop transformations took place so that the validator can have an easier time matching the memory accesses before and after the transformation.

Compiler Independence Presently, our validator validates LLVM 'opt' transformations only. Going forward, we could try to compare a C/C++ program with assembly language. While expanding the language gap between the original and the transformed program will certainly introduce new challenges, such validation would be compiler-independent and would have wider scope of program comparison.

Bibliography

[1] GCC, the GNU compiler collection. http://gcc.gnu.org/.

[2] The Java language specification. https://docs.oracle.com/javase/specs/.

[3] The LLVM compiler infrastructure. http://llvm.org/.

[4] C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[5] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson. Stateless model checking for POWER. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 134–156, 2016. doi: 10.1007/978-3-319-41540-6_8. URL https://doi.org/10.1007/978-3-319-41540-6_8.

[6] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017. doi: 10.1007/s00236-016-0275-0. URL https://doi.org/10.1007/s00236-016-0275-0.

[7] S. V. Adve and M. D. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, pages 2–14, 1990. doi: 10.1145/325164.325100. URL https://doi.org/10.1145/325164.325100.

[8] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi: 10.1145/2627752. URL https://doi.org/10.1145/2627752.

[9] ARM. ARM architecture reference manual (ARMv8, for ARMv8-A architecture profile). https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile, 2017.

[10] D. Aspinall and J. Sevcík. Formalising java's data race free guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 22–37, 2007. doi: 10.1007/978-3-540-74591-4_4. URL https://doi.org/10.1007/978-3-540-74591-4_4.

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 55–66, 2011. doi: 10.1145/1926385.1926394. URL https://doi.org/10.1145/1926385.1926394.

[12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012. doi: 10.1145/2103656.2103717. URL https://doi.org/10.1145/2103656.2103717.

[13] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 283–307, 2015. doi: 10.1007/978-3-662-46669-8_12. URL https://doi.org/10.1007/978-3-662-46669-8_12.

[14] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and opencl. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 634–648, 2016. doi: 10.1145/2837614.2837637. URL https://doi.org/10.1145/2837614.2837637.

[15] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 107–127, 2014. doi: 10.1007/978-3-642-54833-8_7. URL https://doi.org/10.1007/978-3-642-54833-8_7.

[16] H. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 68–78, 2008. doi: 10.1145/1375581.1375591. URL https://doi.org/10.1145/1375581.1375591.

[17] H. Boehm and B. Demsky. Outlawing ghosts: avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC '14, Edinburgh, United Kingdom, June 13, 2014*, pages 7:1–7:6, 2014. doi: 10.1145/2618128.2618134. URL https://doi.org/10.1145/2618128.2618134.

[18] H.-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES 2012, pages 9–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1632-3. doi: 10.1145/2414729.2414732. URL http://doi.acm.org/10.1145/2414729.2414732.

[19] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 104–123, 2010. doi: 10.1007/978-3-642-11970-5_7. URL https://doi.org/10.1007/978-3-642-11970-5_7.

[20] P. Cenciarelli, A. Knapp, and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 331–346, 2007. doi: 10.1007/978-3-540-71316-6_23. URL https://doi.org/10.1007/978-3-540-71316-6_23.

[21] S. Chakraborty. Technical appendix, 2019. Available at http://plv.mpi-sws.org/soham/thesis/index.html.

[22] S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 100–110, 2017. URL http://dl.acm.org/citation.cfm?id=3049844.

[23] S. Chakraborty and V. Vafeiadis. Technical appendix, 2018. Available at http://plv.mpi-sws.org/weakest/.

[24] M. Dodds, M. Batty, and A. Gotsman. C/C++ causal cycles confound compositionality. *TinyToCS*, 2, 2013. URL http://tinytocs.org/vol2/papers/tinytocs2-dodds.pdf.

[25] M. Doko and V. Vafeiadis. A program logic for C11 memory fences. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 413–430, 2016. doi: 10.1007/978-3-662-49122-5_20. URL https://doi.org/10.1007/978-3-662-49122-5_20.

[26] M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 448–475, 2017. doi: 10.1007/978-3-662-54434-1_17. URL https://doi.org/10.1007/978-3-662-54434-1_17.

[27] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the armv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621, 2016. doi: 10.1145/2837614.2837615. URL https://doi.org/10.1145/2837614.2837615.

[28] J. Gosling, B. Joy, and G. L. Steele. The Java language specification, 1996.

[29] ISO/IEC 14882:2011. Programming language C++.

[30] ISO/IEC 9899:2011. Programming language C.

[31] A. Jeffrey and J. Riely. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 759–767, 2016. doi: 10.1145/2933575.2934536. URL https://doi.org/10.1145/2933575.2934536.

[32] J. Kaiser, H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 17:1–17:29, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.17. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.17.

[33] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189, 2017. URL http://dl.acm.org/citation.cfm?id=3009850.

[34] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018. doi: 10.1145/3158105. URL https://doi.org/10.1145/3158105.

[35] O. Lahav and V. Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 479–495, 2016. doi: 10.1007/978-3-319-48989-6_29. URL https://doi.org/10.1007/978-3-319-48989-6_29.

[36] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 649–662, 2016. doi: 10.1145/2837614.2837643. URL https://doi.org/10.1145/2837614.2837643.

[37] O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632, 2017. doi: 10.1145/3062341.3062352. URL https://doi.org/10.1145/3062341.3062352.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439. URL https://doi.org/10.1109/TC.1979.1675439.

[39] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226, 2014. doi: 10.1145/2594291.2594334. URL https://doi.org/10.1145/2594291.2594334.

[40] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[41] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 65–76, 2015. doi: 10.1145/2737924.2737986. URL https://doi.org/10.1145/2737924.2737986.

[42] LLVM Bug #22514. Wrong transformation due to semantic gap between C11 and LLVM semantics. https://llvm.org/bugs/show_bug.cgi?id=22514.

[43] LLVM documentation. LLVM atomic instructions and concurrency guide. http://llvm.org/docs/Atomics.html.

[44] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016. URL http://arxiv.org/abs/1611.01507.

[45] J. Manson, W. Pugh, and S. V. Adve. Causality test cases. http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html, 2004.

[46] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391, 2005. doi: 10.1145/1040305.1040336. URL https://doi.org/10.1145/1040305.1040336.

[47] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, 2012. URL https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf. Draft.

[48] R. Morisset, P. Pawan, and F. Z. Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 187–196, 2013. doi: 10.1145/2491956.2491967. URL https://doi.org/10.1145/2491956.2491967.

[49] K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 304–323, 2013. doi: 10.1007/978-3-642-38856-9_17. URL https://doi.org/10.1007/978-3-642-38856-9_17.

[50] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 83–94, 2000. doi: 10.1145/349299.349314. URL https://doi.org/10.1145/349299.349314.

[51] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 111–128, 2016. doi: 10.1145/2983990.2983997. URL https://doi.org/10.1145/2983990.2983997.

[52] B. Norris and B. Demsky. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10:1–10:51, 2016. doi: 10.1145/2806886. URL https://doi.org/10.1145/2806886.

[53] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 478–503, 2010. doi: 10.1007/978-3-642-14107-2_23. URL https://doi.org/10.1007/978-3-642-14107-2_23.

[54] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009. doi: 10.1007/978-3-642-03359-9_27. URL https://doi.org/10.1007/978-3-642-03359-9_27.

[55] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633, 2016. doi: 10.1145/2837614.2837616. URL https://doi.org/10.1145/2837614.2837616.

[56] J. Y. A. Pichon-Pharabod. *A no-thin-air memory model for programming languages*. PhD thesis, University of Cambridge, 2018. URL https://www.repository.cam.ac.uk/handle/1810/274465.

[57] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pages 151–166, 1998. doi: 10.1007/BFb0054170. URL https://doi.org/10.1007/BFb0054170.

[58] A. Podkopaev, O. Lahav, and V. Vafeiadis. Promising compilation to ARMv8 POP. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 22:1–22:28, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.22.

[59] A. Podkopaev, O. Lahav, and V. Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *PACMPL*, 3(POPL):69:1–69:31, 2019. doi: 10.1145/3290382. URL https://doi.org/10.1145/3290382.

[60] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000. doi: 10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A. URL https://doi.org/10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A.

[61] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018. doi: 10.1145/3158107. URL https://doi.org/10.1145/3158107.

[62] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391. ACM, 2009. doi: 10.1145/1480881.1480929. URL https://doi.org/10.1145/1480881.1480929.

[63] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011. doi: 10.1145/1993498.1993520. URL https://doi.org/10.1145/1993498.1993520.

[64] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322. ACM, 2012. doi: 10.1145/2254064.2254102. URL https://doi.org/10.1145/2254064.2254102.

[65] J. Sevcík. *Program transformations in weak memory models*. PhD thesis, University of Edinburgh, UK, 2009. URL http://hdl.handle.net/1842/3132.

[66] J. Sevcík. Safe optimisations for shared-memory concurrent programs. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 306–316. ACM, 2011. doi: 10.1145/1993498.1993534. URL https://doi.org/10.1145/1993498.1993534.

[67] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013. doi: 10.1145/2487241.2487248. URL https://doi.org/10.1145/2487241.2487248.

[68] A. Sezgin. *Formalization and Verification of Shared Memory*. PhD thesis, The University of Utah, 2004. URL http://formalverification.cs.utah.edu/dissertations/phd/sezgin_formalizationandverificationofsharedmemory_2004diss.pdf.

[69] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 391–406. ACM, 2013. doi: 10.1145/2509136.2509509. URL https://doi.org/10.1145/2509136.2509509.

[70] K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 357–384. Springer, 2018. doi: 10.1007/978-3-319-89884-1_13. URL https://doi.org/10.1007/978-3-319-89884-1_13.

[71] J. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 295–305. ACM, 2011. doi: 10.1145/1993498.1993533. URL https://doi.org/10.1145/1993498.1993533.

[72] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707. ACM, 2014. doi: 10.1145/2660193.2660243. URL https://doi.org/10.1145/2660193.2660243.

[73] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884. ACM, 2013. doi: 10.1145/2509136.2509532. URL https://doi.org/10.1145/2509136.2509532.

[74] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220. ACM, 2015. doi: 10.1145/2676726.2676995. URL https://doi.org/10.1145/2676726.2676995.

[75] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986. doi: 10.1007/3-540-17906-2_31. URL https://doi.org/10.1007/3-540-17906-2_31.

[76] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi: 10.1145/1993498.1993532. URL https://doi.org/10.1145/1993498.1993532.

11 Curriculum vitae

Research Interests

Concurrency, Relaxed Memory Consistency, Compilation

Current Position

PhD [November 2013–]
Software Analysis and Verification Group
Max Planck Institute for Software Systems, Germany
Thesis advisor: Dr. Viktor Vafeiadis

Education

Master of Science (MS) [2005–2008]
Department of Computer Science and Engineering
Indian Institute of Technology (IIT) Kharagpur, India

Bachelor of Engineering (BE) [2000–2004]
Computer Science and Engineering
Vidyasagar University

Awards and Honors

- Invited to the Dagstuhl seminar *Program Equivalence* 2018.
- Recipient, ACM SIGPLAN PAC funding to attend CGO 2017.
- AMD Geo Excellence Award 2013.

Industrial R&D Experience

AMD Compiler Group India [July 2011 – September 2013]

- Enablement of LLVM compiler for HSA systems.
- Vectorization and code generation in LLVM.

- Vectorization in Open64 compiler.
- Performance analysis of SPEC CPU2006 benchmarks.

TCS Research (TRDDC) India [February 2010 – June 2011]

- Efficient test execution plan generation for software test cycles.

IBM Research Lab India [March 2008 - February 2010]

- Bug detection tool for SAP ABAP programs
- Software model to model transformations
- Analysis and refactoring for parallelization.

Teaching Experience

Teaching Assistantships

- Program Synthesis Seminar, Summer 2018.
- Software Engineering undergraduate course, Spring 2007.
- Object Oriented System Implementation, Autumn, 2006.
- Programming and Data Structure Lab, Spring 2006.
- Programming and Data Structure Lab, Spring 2005.

Courseware Development

- Weak Memory Consistency, Summer 2017.
- Object-Oriented (C#/.NET centric) Courseware Development, 2004–2007

Talks

- Validating optimizations of concurrent C/C++ programs.
  Dagstuhl seminar *Program Equivalence* 2018.
- Formalizing the concurrency semantics of an LLVM fragment.
  Aarhus Concurrency Workshop 2017.
- Formalizing the concurrency semantics of an LLVM fragment.
  EuroLLVM 2017.
- Formalizing the concurrency semantics of an LLVM fragment.
  CGO 2017.
- Validating optimizations of concurrent C/C++ programs.
  CGO 2016.

Service

- Reviewer: ESOP 2017, FoSSaCS 2019.
- Artifact Evaluation Committee (AEC): CGO 2018, PLDI 2018.

Recent Publications

- Grounding thin-air reads with event structures. Soham Chakraborty, Viktor Vafeiadis. In POPL 2019.
- Formalizing the concurrency semantics of an LLVM fragment. Soham Chakraborty, Viktor Vafeiadis. In CGO 2017.
- Validating optimizations of concurrent C/C++ programs. Soham Chakraborty, Viktor Vafeiadis. In CGO 2016.
- Improved MHP analyses. Aravind Sankar, Soham Chakraborty, V. Krishna Nandivada. In CC 2016.
- Common compiler optimisations are invalid in the C11 memory model and what we can do about it. Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, Francesco Zappa Nardelli. In POPL 2015.
- Aspect-oriented linearizability proofs. Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, Viktor Vafeiadis. In LMCS (11) 2015.

Patents

- Automated test cycle estimation system and method. US9032370B2. Soham Sundar Chakraborty, Pavan Kumar Chittimalli, Vipul Shah.
- Scalable partial vectorization. US9158511B2. Ramshankar Ramanarayanan, Meghana Gupta, Soham S. Chakraborty, Dibyendu Das.
- Automated test execution plan derivation system and method. US US9378120B2. Soham Sundar Chakraborty, Vipul Shah.