Ph.D. DISSERTATION

Understanding and Fulfilling the Desiderata for Relaxed Memory Models

느슨한 메모리 모델이 갖추어야 하는 성질을 이해하고 실현하기

August 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Sung-Hwan Lee

Understanding and Fulfilling the Desiderata for Relaxed Memory Models

느슨한 메모리 모델이 갖추어야 하는 성질을 이해하고 실현하기

지도교수 허 충 길

이 논문을 공학박사학위논문으로 제출함

2023 년 7 월

서울대학교 대학원

컴퓨터공학부

이 성 환

이성환의 공학박사 학위논문을 인준함

2023 년 7 월

위 원 장	이광근	(인)
부위원장	허충길	(인)
위 원	Ori Lahav	(인)
위 원	강지훈	(인)
위 원	김지응	(인)

Abstract

Defining a relaxed memory model has been a major challenge for a couple of decades. The challenge stems from a sharp conflict between the two desiderata for relaxed memory models, usability and efficiency. Usability requires a model to be sensible and understandable, allowing programmers to use the model for developing and reasoning about concurrent programs. Efficiency requires a model to be efficiently implementable, *i.e.*, it validates common compiler transformations and can be efficiently mapped to hardware. While the two desiderata are the key principles in designing a relaxed memory model, a significant challenge exists in balancing between them.

This dissertation thoroughly understands the desiderata for relaxed memory models, identifies inherent conflicts between them, and resolves the conflicts at a minimal price. The first part of this dissertation presents PS 2.0, the first relaxed memory model that provably supports compiler transformations based on global analyses. PS 2.0 redesigns the key components of the promising semantics (PS) by Kang et al. and validates global value optimization and register promotion, while preserving the known results for PS, such as data-race-freedom guarantees. PS 2.0 also resolves the problem of the inefficiency in compiling read-modify-write (RMW) operations of PS to Armv8, which requires an unintended extra fence. The second part investigates the problem of developing an in-order memory model that executes instructions following their program order, placing emphasis on amenability to reasoning and verification. We demonstrate that an in-order model can validate all the compiler optimizations performed on single-threaded code by utilizing the distinction between an in-order source model and an out-of-order intermediate representation model. For atomics, we propose a pragmatic solution for mapping relaxed writes, for which reordering with previous reads should be prevented, with negligible performance overhead.

Keywords: concurrency, relaxed memory models, operational semantics, compiler optimizations, formal methods
Student Number: 2017-23151

Contents

Chapter I

Introduction

Concurrency is crucial in modern software for fully utilizing multiple cores or processors. Although most programmers avoid data races in their programs (*e.g.*, by using locks), concurrent programming with benign races is required for implementing non-blocking algorithms or synchronization primitives. The problem is that such programs cannot assume simple semantics such as interleaving execution or sequential consistency (SC) due to *relaxed behaviors* arising from optimizations performed by compilers and hardware. While the effect of these optimizations are invisible in single-threaded programs (imagine the reordering of two independent memory accesses), they become visible in concurrent programs where multiple threads access a shared memory location at the same time. A *relaxed memory model* (or a weak memory model) is the semantics of concurrent programs describing the relaxed behaviors in an abstract way.

Defining a relaxed memory model has been a major challenge in programming languages for a couple of decades. The challenge stems from a tension between the two desiderata for relaxed memory models, efficiency and usability. Efficiency requires a memory model to be efficiently implementable meaning that the model should validate common compiler transformations and can be efficiently mapped to hardware architectures. Usability allows programmers to understand and use a memory model

for programming and reasoning. For this, a model should admit reasoning principles such as data-race-freedom (DRF) guarantees, and it should be simple enough for broad programmers to understand. While the two desiderata are the key principles in designing a relaxed memory model, it has been known that there is a significant challenge in balancing between them [7, 65, 33, 44, 17, 42, 37, 26, 55].

One of the challenges in defining a relaxed memory model has been to properly capture *load buffering* (LB) behavior:

$$\begin{array}{l|l} a := X^{\texttt{rlx}} \;\; /\!/ 1 & b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ Y^{\texttt{rlx}} := 1 & X^{\texttt{rlx}} := b \end{array} \tag{LB}$$

Here, a and b are thread-local variables, and X and Y are shared memory locations initialized to 0. We also assume that all memory accesses here are relaxed atomics of C/C++ (*i.e.*, `rlx`). Then, the annotated outcome $a = b = 1$ can be observed once the two instructions of the first thread are reordered by compilers or hardware. Indeed, such reordering is allowed by architectures like Arm and Power, and LB is actually observed on some Arm CPUs.

The problem is that the C/C++ memory model allows the same behavior for the following variant of the LB program above:

$$\begin{array}{l|l} a := X^{\texttt{rlx}} \;\; /\!/ 1 & b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ Y^{\texttt{rlx}} := a & X^{\texttt{rlx}} := b \end{array} \tag{OOTA}$$

As every thread simply copies the value it reads, both threads are not supposed to read any other value than 0 from the memory. However, the annotated behavior, often called *out-of-thin-air*, is allowed in C11 [8], breaking the basic invariant-based reasoning above. To solve this problem, multiple solutions [33, 15, 55, 26, 29] has been proposed. However, these models employ complex mechanisms for distinguishing LB example from OOTA example. Moreover, the models often adopt different guiding principles for their design, overlooking other requirements for relaxed memory models.

This dissertation thoroughly understands the desiderata for relaxed memory models in terms of efficiency and usability, identifies inherent conflicts between them, and proposes memory models that resolve the conflicts at a minimal price. Specifically, the first part of this dissertation focuses on efficiency and presents the first relaxed memory model that provably supports compiler transformations based on certain

analyses that analyze multiple threads. In the second part, we observe that an in-order memory model that executes instructions simply following the program code can be implemented with only a negligible performance overhead. Note that all the main results of this dissertation are fully mechanized in the Coq proof assistant [25]. The supplementary material for this dissertation including the mechanized proofs is available online [1].

In the following, we summarize the two parts of this dissertation.

Inter-thread optimizations in relaxed memory concurrency (Chapter III). Despite many years of research, there had been no relaxed memory model that properly supports inter-thread optimizations, namely compiler transformations whose validity depend on some compiler analyses that analyze multiple threads (*e.g.*, global analyses that analyze the whole program). Some examples of such transformations include (*i*) removal of null pointer or array bound checks based on a global value analysis; and (*ii*) register promotion that converts shared memory variables that happen to be accessed by only one thread to thread-local variables. The latter is particularly useful when thread-safe libraries are used by a single thread, and it is also crucial in languages like Java that have only atomic accesses, However, supporting inter-thread optimizations in relaxed memory models is challenging due to a subtle interaction between inter-thread and intra-thread (or simply, "local") optimizations. For instance, a local optimization, such as dead store elimination, may change a global analysis result, thereby enabling a new inter-thread optimization that was invalid before the local optimization. Then, this inter-thread optimization may further enable other intra- or inter-thread optimizations. Therefore, a naive approach, such as incorporating global analyses in the memory model, cannot fully support intra- and inter-thread transformations.

To address this challenge, we developed the promising semantics 2.0 model (PS2) that *provably supports the above-mentioned inter-thread optimizations*, without losing any thread-local optimizations allowed for the original PS [33]. PS2 made two major changes to PS. First, it redesigned the key components of the original PS model, promises and certification, to validate compiler transformations based on global value-range analysis. Second, to support register promotion, PS2 introduced reservations,

which allow a thread to reserve a memory location so it can exclusively perform an atomic read-modify-write (RMW) operation to that location. The reservations also resolve the problem with the sub-optimal compilation of PS to Armv8 that required an unintended fence for mapping relaxed RMWs.

This chapter contains the text and figures of the following paper:

[44] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI 2020*.

An in-order semantics for relaxed memory concurrency (Chapter IV). Recent proposals of relaxed memory models employ an out-of-order semantics for efficiency, in particular, to support *load-store reordering* that is allowed by modern hardware architectures. On the contrary, an in-order model, which executes a program following the order of instructions in the program code, is more suitable for reasoning and verification. Indeed, verification for an in-order weak memory model RC11 [37] has been extensively studied, and multiple tools and techniques, such as program logics and model checkers, have been developed.

In this chapter, we investigate the problem of developing an in-order semantics for relaxed memory concurrency at a minimal cost. As a result, we propose the first solution that *fully supports non-atomic accesses* without any performance overhead while assuming an *in-order source semantics*. In particular, our model validates all compiler optimizations on non-atomics performed in single-threaded code, including load introduction, which is notoriously difficult to support in an in-order model. The key idea in this approach is to *split* the semantics into two models: (*i*) vRC11, an in-order source model accounting for reasoning and verification; and (*ii*) PS^{IR}, an out-of-order intermediate representation (IR) model accounting for compiler transformations. Crucially, we formally proved the soundness of the (identity) mapping from vRC11 to PS^{IR}. Roughly speaking, this is possible because vRC11 adopts a *catch-fire* semantics (as the C/C++11 memory model and RC11 do) that invokes an *undefined behavior* for data races. Then, the catch-fire of vRC11 sufficiently accounts for any program behavior allowed by an out-of-order execution under PS^{IR}. Notably, vRC11 is proven to be stronger than RC11, and thus all the verification tools and techniques

developed for RC11 can also be applied to verification under vRC11. Note that the IR model PS^{IR} is based on PS2, and thus, PS^{IR} supports inter-thread optimizations studied in the first part.

Unfortunately, for atomic accesses, we proved that any in-order source model that supports all optimizations on non-atomics could *never* allow the reordering of a non-atomic/relaxed read followed by a relaxed write. Nevertheless, existing compilers do not perform this kind of optimization. Still, this is allowed by mainstream hardware architectures. Therefore, we proposed a pragmatic solution for mapping PS^{IR} to hardware architectures that would entail almost no performance overhead. Specifically, we found that load-store reordering is observed only on a few hardware implementations and learned from hardware engineers that the performance impact of load-store reordering is rather limited in the Arm CPU design. Accordingly, we suggested introducing a new kind of store instructions called *strong stores* that is never reordered with previous load instructions. Then, a relaxed write in PS^{IR} can be soundly mapped to a strong store of hardware. We formally verified the soundness of mapping PS^{IR} to the operational model for Armv8 [59] extended with the strong stores.

This chapter contains the text and figures of the following paper:

[42] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, Ori Lahav. Putting Weak Memory in Order via a Promising Intermediate Representation. In *PLDI 2023*.

Outline. The rest of this dissertation is structured as follows. Chapter II introduces preliminaries including the promising semantics by Kang et al. [33] and data-race-freedom guarantees for concurrent programs. Chapter III and Chapter IV presents the two parts as described above. Finally, Chapter V concludes this dissertation and discusses some future directions.

Chapter II

Background

1 The Promising Semantics

In this section, we introduce the promising semantics (PS) of Kang et al. [33]. For simplicity, we present only a fragment of PS containing only three kinds of memory accesses: *relaxed* (the default mode), *release writes* (`rel`), and *acquire reads* (`acq`). Read-modify-write (RMW) instructions, such as compare-and-swap (`CAS`) and fetch-and-add (`FADD`), carry two access modes—one for the exclusive read and one for the write. We put aside other access modes, fences, and release sequences, as they are orthogonal to the contribution of this paper. We refer the reader to [33] for the full PS model.

Domains We assume non-empty sets Loc of locations and Val of values. We also assume a set Time of *timestamps*, which is totally and densely ordered by $<$ with 0 as its minimum. (In our examples, we take non-negative rational numbers as timestamps with their usual ordering.) A *view*, $V \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$, records the largest known timestamp for each memory location. A timestamp *interval* is a pair of timestamps $(f, t]$ with $f < t$ or $f = t = 0$. It represents the range of timestamps from (but not including) f up to and including t.

Memory In PS, the memory is a set of *messages* representing all previously executed writes. A message m is of the form $\langle X@(f,t], v, V_{\mathsf{m}}\rangle$, where $X \in \mathsf{Loc}$ is the location, $v \in \mathsf{Val}$ is the stored value, $(f,t]$ is a timestamp interval, and $V_{\mathsf{m}} \in \mathsf{View}$ is the message view. The latter is used to model release-acquire synchronization and will be explained shortly. Initially, the memory consists of an initialization message for every location carrying the value 0, the interval $(0,0]$, and the bottom view $\bot \triangleq \lambda X.\ 0$. We require that any two messages with the same location in memory have disjoint timestamp intervals. The timestamp (also called the *"to"-timestamp*) of a massage $\langle X@(f,t], v, V_{\mathsf{m}}\rangle$ is the upper bound t of the message's timestamp interval. The lower bound f, called the *"from"-timestamp*, is used to model the atomicity of RMW operations as explained below.

Machine State PS is an operational model where threads execute in an interleaved fashion. The *machine state* is a pair $\Sigma = \langle \mathcal{T}, M\rangle$, where $\mathcal{T}$ assigns a *thread state* T to every thread and M is a (global) *memory*. A thread state is a triple $T = \langle \sigma, V, P\rangle$ where σ is the local store recording the values of its local variables, $V \in \mathsf{View}$ is the *thread view*, and P is a set of messages representing the thread's outstanding promises.

Relaxed Reads and Writes Thread views are instrumental in providing correct semantics to memory accesses. The thread view, V, records the "knowledge" of each thread, *i.e.*, the timestamp of the most recent message that it has observed for each location. It is used to forbid a thread to read from a (stale) message m if the thread is aware of a "newer" message, *i.e.*, when $V(X)$ is greater than the message's timestamp. Similarly, when a thread adds messages of location X to the memory, it has to pick a timestamp t for the added message that is greater than its view of X ($V(X) < t$):

READ. A thread can read from memory M by simply observing a message $\langle X@(f,t], v, \_\rangle \in M$ provided that $V(X) \le t$, and updating its view for X to t.

WRITE. A thread adds a new message $m = \langle X@(f,t], v, \bot\rangle$ to the memory where the timestamp t is greater than the thread's view of X ($V(X) < t$) and there is no other message with the same location and overlapping timestamp interval in the memory. Relaxed writes set the message view to $\bot$, which maps each location to timestamp 0.

The following example illustrates how timestamps of messages and views interact.

Note that we assume that both threads start with the initial thread view that maps X and Y to 0, and that every location is initialized to 0: the initial memory only contains messages $\langle X@(0,0], 0, \bot\rangle$ and $\langle Y@(0,0], 0, \bot\rangle$.[1]

$$\begin{array}{l} X^{\texttt{rlx}} := 1 \\ a := Y^{\texttt{rlx}} \;\; /\!/ 0 \end{array} \;\Big\|\; \begin{array}{l} Y^{\texttt{rlx}} := 1 \\ b := X^{\texttt{rlx}} \;\; /\!/ 0 \end{array} \tag{SB}$$

Here, both threads are allowed to read from the initialization messages, 0. When thread 1 performs the write to X, it will add a message $\langle X@(f,t], 0, \bot\rangle$ by choosing some $t > f \geq 0$. During this write, thread 1 should increase its view of X to t, while maintaining $V(Y)$ to be 0 as it was. Hence, thread 1 is still allowed to read 0 from y in the subsequent execution. As thread 2 can be executed in the same way, both threads are allowed to read 0.

Relaxed RMWs Read-modify-write (RMW) operations are essentially a pair of accesses to the same location—a read followed by a write—with an additional atomicity guarantee: the read reads from a message that immediately precedes the one added by the write. PS employs timestamp intervals (rather than single timestamps) to enforce atomicity.

UPDATE. When a thread performs an RMW, it first reads a message $\langle X@(f,t], v, \bot\rangle$, and then writes the updated message with "from"-timestamp equal to t, *i.e.*, a message of the form $\langle X@(t,t'], v', \bot\rangle$. This results in consecutive messages $(f,t], (t,t']$, forbidding other writes to be later placed between the two messages (recall that messages with the same location must have disjoint timestamp intervals).

This constraint, in particular, means that two competing RMWs cannot read from the same message, as the following "parallel increment" example demonstrates.[2]

$$a := \texttt{FADD}(X, 1) \;\; /\!/ 0 \;\Big\|\; b := \texttt{FADD}(X, 1) \;\; /\!/ 0 \tag{Upd}$$

Without loss of generality, suppose that thread 1 executed first. As it performs an RMW operation, it must "attach" the message it adds to an existing message. Since

[1] In all our code examples, we assume that all memory accesses are relaxed (`rlx` memory order) unless annotated otherwise.

[2] Here and henceforth, we assume that RMW instructions such as `FADD` and `CAS` return the value that was read during the read-modify-write operation (before the update).

the only existing message in this stage is the initial one $\langle X@(0,0], 0, \bot\rangle$, thread 1 will first add a message $m = \langle X@(0,t], 1, \bot\rangle$ with some $t > 0$ to the memory. Then, the RMW of thread 2 cannot also read from the initial message because its interval would overlap with the $(0,t]$ interval of m. Therefore, the annotated behavior is forbidden. More abstractly speaking, the timestamps intervals of PS express a dense total order on messages to the same location together with immediate adjacency constraints on this order, which are required for handling RMW operations.

Release and Acquire Accesses To provide the appropriate semantics to release and acquire accesses, PS uses the message views. Indeed, a release write should transfer the current knowledge of the thread to other threads that read the message by an acquire read. Thus, (*i*) a release write operation puts the current thread view in the message view of the added message; and (*ii*) an acquire read operation incorporates the view of the message being read in the thread view (by taking the pointwise maximum).

READ is defined the same as before, except that when the thread performs an *acquire* read, it increases its view to contain not only the ("to") timestamp of the message read but also the view of that message.

WRITE is defined as before, except that *release* writes record the thread view in the message being added, whereas relaxed writes record the $\bot$ view.

As a result, the acquiring thread is confined in its future reads at least as the releasing thread was confined when it "released" the message being "acquired". As a simple example, consider the following:

$$\begin{array}{l} X^{\texttt{rlx}} := 1 \\ Y^{\texttt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := Y^{\texttt{acq}} \;\; /\!/ 1 \\ \texttt{if}\ a = 1\ \texttt{then} \\ \quad b := X^{\texttt{rlx}} \;\; /\!/ 0 \end{array} \tag{MP}$$

Here, if thread 2 reads 1 from Y, which is written by thread 1, both threads are synchronized through release and acquire. Thus, thread 2 obtains the knowledge of thread 1, namely its view for X is increased to include the timestamp of $X^{\texttt{rlx}} := 1$ of thread 1. Therefore, after reading 1 from Y, thread 2 is not allowed to read the initial value 0 from X.

Release/acquire RMW operations also transfer thread views via message views as release writes and acquire reads do.

Promises The main novelty of PS lies in its way to enable the reordering of a read followed by a write (of different locations), needed to explain the outcome of the LB program in §11. Thus, besides step-by-step program execution, PS allows threads to non-deterministically *promise* their future writes. This is done by simply adding a message (whose interval does not overlap with that of any existing message to the same location) to the memory. Later, the execution of write instructions may also *fulfill* an existing promise (rather than add a message to the memory). Thread promises are kept in the thread state, and removed when the promise is fulfilled. Naturally, at the end of the execution all promises must be fulfilled.

PROMISE. At any point, a thread can add a message to both its set of promises and the memory.

FULFILL. A thread can fulfill its promise by executing a (non-release) write instruction, by removing a message from the thread's set of promises. PS does not allow release writes to be promised, *i.e.*, a promise cannot be fulfilled through a release write instruction.

In the LB program above, thread 1 may promise $Y^{\texttt{rlx}} := 1$ at first. This allows thread 2 to read 1 from Y and write it back to X. Then, thread 1 can read 1 from X, which was written by thread 2, and fulfill its promise.

Certification To ensure that promises do not make the semantics overly weak, each sequence of steps by a thread (before "yielding control to the scheduler") has to be *certified*: the thread that took the steps should be able to fulfill all its promises when executed in isolation. Indeed, revisiting the LB program above, note that at the point of promising $Y^{\texttt{rlx}} := 1$ (in the very beginning of the run), thread 1 can run and perform $Y^{\texttt{rlx}} := 1$ without any "help" of other threads.

Certification (*i.e.*, the thread-local run fulfilling all outstanding promises of the thread) is necessary to avoid "thin-air reads" as demonstrated by the LB program given in Chapter I. As every thread simply copies the value it reads, both threads are not supposed to read any other value than 0 from the memory. However, the annotated behavior, often called out-of-thin-air, is allowed in C11 [8]. In PS, if a thread could promise without certification, this behavior would be allowed by the same execution as the one for LB. However, with the certification requirement, thread 1 cannot promise

$Y^{\texttt{rlx}} := 1$, as, when running in isolation, thread 1 will only write $Y^{\texttt{rlx}} := 0$.

PS requires a certification to exist for *every future memory* (*i.e.*, any memory that extends the current memory). In §4, we explain the reason for this condition and its consequences.

Machine Step A *thread configuration* $\langle T, M\rangle$ can take one of **READ**, **WRITE**, **UPDATE**, **PROMISE**, and **FULFILL** steps, denoted by $\langle T, M\rangle \to \langle T', M'\rangle$. In addition, a thread configuration is called *consistent* if for every future memory M_{future} of M, there exist T' and M' such that (where $T.\texttt{prm}$ denotes the set of outstanding promises in thread state T):

$$\langle T, M_{\text{future}}\rangle \to^* \langle T', M'\rangle \quad \wedge \quad T'.\texttt{prm} = \emptyset$$

In turn, the machine step is defined as follows:

$$\frac{\langle \mathcal{T}(\tau), M\rangle \to^+ \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ is consistent}}{\langle \mathcal{T}, M\rangle \to \langle \mathcal{T}[\tau \mapsto T'], M'\rangle}$$

We note that the machine step is completely *thread-local*: it is only determined by the local state of the executing thread and the global memory, independently of the other threads' states. Thread-locality is a key design principle of PS. It is what makes PS conceptually well-behaved, and, technically speaking, it allows one to prove the validity of various local program transformations, which are performed by compilers and/or hardware, using standard thread-local simulation arguments.

To show a concrete example, we list the execution steps of PS leading to the annotated behavior of the LB program (items prefixed with "C" represent certification steps):

(1) Thread 1 promises $\langle Y@(1, 2], 1, \bot\rangle$.

  (C1) Starting from an arbitrary extension of the current memory, thread 1 reads $\langle X@(0, 0], 0, \bot\rangle$, the initial message of X.

  (C2) Thread 1 fulfills its promise $\langle X@(1, 2], 1, \bot\rangle$.

(2) Thread 2 reads $\langle Y@(1, 2], 1, \bot\rangle$.

(3) Thread 2 writes $\langle X@(1,2], 1, \bot\rangle$.

(4) Thread 1 reads $\langle X@(1,2], 1, \bot\rangle$.

(C1) Starting from an arbitrary extension of the current memory, Thread 1 fulfills its promise $\langle Y@(1,2], 1, \bot\rangle$.

(5) Thread 1 fulfills its promise $\langle Y@(1,2], 1, \bot\rangle$.

2 Data-race-freedom Guarantees

Data-race-freedom guarantees are the fundamental programmability guarantees that ensure a simple and strong semantics such as sequential consistency (SC) for programs that avoid data-races. For example, DRF-SC ensures that data-race-free programs only exhibit behaviors allowed under SC semantics. Here, it is crucial that the data-race-freedom can be checked under the strong semantics SC, so the clients of DRF-SC are only required to understand SC semantics but not the underlying complext weak memory model. Therefore, DRF guarantees make concurrent programming easier and more accessible by allowing most programmers who avoid races in their programs to assume strong semantics, such as SC, instead of understanding the full complextiy of weak memory.

Recent studies have proposed more refined DRF guarantees that ensure a weaker semantics such as release/acquire (RA) given a relaxed notion of data-race-freedom premise [15, 33]. In particular, DRF-RA guarantee ensures RA semantics (*i.e.*, executing every read as an acquire read and every write as an release write) for programs that, under RA semantics, have no data race involving accesses weaker than release and acquire accesses (*e.g.*, relaxed accesses).

In the context of the promising semantics, data races can be naturally defined as states in which two different threads can access the same location and at least one of these accesses is writing. Then, for example, by analyzing the MP example under RA semantics, one can easily observe that the only race is on the `rel`/`acq` accesses to Y. (Importantly, such analysis safely ignores promises, since promises are not allowed under RA.) Then, DRF-RA implies that MP has only RA behaviors. In contrast, in the

LB example, non-RA behaviors are possible, and, indeed, under RA semantics, there are races on relaxed accesses (to both X and Y).

Another important DRF property for PS is DRF-PF that ensures promise-free (PF) semantics, obtained by forbidding PROMISE step, for programs without data races involving *promisible* writes. In Chapter IV, we observe that the proof of the soundness of mapping the in-order source model based on PF to the out-of-order IR model based on PS is essentially similar to the proof of DRF-PF. DRF-PF also serves as a building block for proving DRF-RA for PS.

Lately, a *modular* notion of DRF guarantees have been proposed [24] and established for PS [17]. These DRF theorems, often called "local" DRF (LDRF), are proposed to overcome a global nature of traditional DRF guarantees that are only applicable when the whole program avoids data races. In contrary, LDRF ensures strong semantics for accesses to a certain set of memory locations $\mathcal{L}$ when the program has no race between accesses to $\mathcal{L}$ assuming the strong semantics for accesses to $\mathcal{L}$. For establishing LDRF guarantees for PS, Cho et al. [17] adopts a bit different notion of data races. Specifically, an access to a location L by a thread τ is considered racy if there is a message to L with the timestamp higher than τ's view of L. In Chapter IV, we adopt a similar notion for defining data races between non-atomic accesses.

In Chapter III, DRF-RA provides us with the main guideline for making sure that our semantics is not overly weak (*i.e.*, we exclude any semantics that breaks DRF-RA). DRF-RA also serves as a main step towards "DRF-Lock", which states that properly locked programs have only sequentially consistent semantics.[3]

[3]The more standard DRF-SC is not applicable here since PS lacks SC accesses. The extension of PS with SC accesses is left to future work.

Chapter III

Inter-thread Optimizations in Relaxed Memory Concurrency

3 Introduction

While there are multiple partial solutions to the problem of defining a proper semantics for relaxed memory concurrency [33, 15, 27, 48, 56], none of them properly supports inter-thread optimizations, namely program transformations whose validity depend on some analyses that analyze multiple threads. Examples of such transformations are (*i*) removal of null pointer checks based on global null-pointer analysis; (*ii*) removal of array bounds checks based on global size analysis; and (*iii*) *register promotion*, *i.e.*, converting accesses to a shared variable that happens to be used by only one thread to local accesses. The latter is particularly useful when thread-safe libraries are used by a single thread, and it is also crucial in languages like Java that have only atomic accesses, but is also useful for C/C++. For instance, in single-threaded programs, it allows the removal of locks, as well as the promotion to register accesses of inlined function calls of concurrent data-structures.

The desire to support inter-thread optimizations in concurrent programming languages goes at least as back as 15 years ago with the Java memory model (JMM) [48]. In fact, the very first JMM "causality test case" is centered around value-range analysis.

Assuming all variables are initialized to 0, JMM allows the annotated outcome of the following example:

$$\begin{array}{l} a := X^{\texttt{rlx}} \;\; /\!/ 1 \\ \texttt{if } a \geq 0 \texttt{ then} \\ \quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{JMM1}$$

> *"Decision: Allowed, since interthread compiler analysis could determine that X and Y are always non-negative, allowing simplification of $a \geq 0$ to true, and allowing write $Y^{\texttt{rlx}} := 1$ to be moved early."* [31]

Supporting inter-thread optimizations, however, is rather challenging because of their interaction with intra-thread (or local) transformations. Inter-thread optimizations generally depend on invariants deduced by some global analyses but these invariants need not hold in the source program; they might hold after some local transformations have been applied. In the following example, (only) after the local elimination of the overwritten $X^{\texttt{rlx}} := 42$ assignment, the condition $a < 10$ becomes a global invariant, and so can be simplified to true as in JMM1.

$$\begin{array}{l} a := X^{\texttt{rlx}} \;\; /\!/ 1 \\ \texttt{if } a < 10 \texttt{ then} \\ \quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} X^{\texttt{rlx}} := 42 \\ b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{LB-G}$$

In more complex cases, an inter-thread optimization may enable a local transformation, which may further enable another inter-thread optimization, which may enable another local optimization, and so on. As a result, supporting both intra- and inter-thread transformations is very difficult, and none of the solutions so far has managed to fully support global analyses along with all the expected intra-thread transformations.

In this chapter, we present the first memory model that solves this challenge: (*i*) it allows the aforementioned inter-thread optimizations (value-range analysis and register promotion); (*ii*) it validates the thread-local compiler optimizations that are validated by the C/C++11 model [37] (*e.g.*, roach-motel reorderings [67]); (*iii*) it can be efficiently mapped to the mainstream hardware platforms (x86, POWER, Armv7, Armv8, RISC-V); and (*iv*) it supports reasoning principles in the form of DRF guarantees, allowing programmers to resort to simpler well-behaved models when data races are appropriately restricted. In developing our model we mainly use (i)–(iii) to

conclude that some behavior should be allowed; while (iv) tells us which behaviors must be forbidden.

As a starting point, we take the *promising semantics* (PS) of Kang et al. [33], a concurrency semantics that satisfies *almost* all our desiderata. It supports almost all C/C++11 features, all expected thread-local compiler optimizations, and several DRF theorems. In addition, Podkopaev et al. [57] established the correctness of a mapping from PS to hardware.[1] The main drawback of PS is that it does not support inter-thread optimizations.

PS is an operational semantics which represents shared memory as a set of messages (*i.e.*, writes). To support out-of-order execution, PS employs a non-standard step, allowing a thread to *promise* to perform a write in the future, which enables other threads to read from it before the write is actually executed.

The technical challenge resides in identifying the exact conditions on such promise steps so that basic guarantees (like DRF and no "thin-air values") are maintained.

In PS, these conditions are completely thread-local: the thread performing the promise must be able to run in isolation from *all extensions* of the current state and fulfill all its outstanding promises. While thread-locality is useful, quantifying over all extensions of the current state prevents optimizations based on global analyses because some extensions may well not satisfy the invariant produced by the analysis.

Checking for promise fulfillment only from the current state without extension enables global analysis, but breaks the DRF guarantee (see §5). Our solution is therefore to check promise fulfillment for a carefully crafted extension of the current state, which we call *capped memory*. Because capped memory does not contain any new values, it is consistent with optimizations based on global value analysis. However, it still does not allow optimizations like register promotion.

To support register promotion, we introduce *reservations*, which allow a thread to secure an exclusive right to perform an atomic read-modify-write instruction reading from a certain message without fixing the value that it will write (because, for example, that might not have yet been resolved). In addition, reservations resolve a problem with the compilation of PS to Armv8, whose intended mapping of RMWs was unsound

[1] Albeit, the mapping of RMWs to Armv8 contains one more barrier ("ld fence") than intended because the intended mapping is unsound.

and required an extra fence [57].[2]

With these two new concepts, we are able to retain the thread-local nature of PS and yet fully support inter-thread optimizations and the intended mapping of RMWs along with all the results available for PS. Our redesigned PS2 model is the first weak memory model that achieves these results. To establish confidence in our model, we have formalized our key results in the **Coq proof assistant**.

Outline In the rest of this chapter, we first review why the PS does not support inter-thread optimizations (§4). We then present our PS2 model both informally in an incremental fashion (§5) and formally all together (§6). In §7, we establish the correctness of mappings from PS2 to hardware, and show that PS2 supports all the local transformations and reasoning principles known to be allowed by PS, as well as register promotion, and the introduction of 'assert' statements for invariants derived by a global analysis. The mechanization of our main results in Coq is available in [43].

4 Problem Overview

As we will shortly demonstrate, the main challenge in PS is to come up with an appropriate thread-local condition for certifying the promises made by a thread. Maintaining thread-locality is instrumental in proving correctness of many compiler transformations, but is difficult to achieve given that promises of different threads may interact.

As we briefly mentioned above, PS requires a certification to exist for any memory that extends the current memory. We start by explaining why certifying promises only from the current memory (without quantifying over all future memories) is not good enough. Kang et al. [33] observed that such model may deadlock: the promising thread may fail to fulfill its promise since the memory was changed since the promise was made. In this work, we observe that a model that requires certifying promises only from the current memory has much more severe consequences. It actually *breaks*

[2]Our current mechanized proof requires a fake control dependency from relaxed fetch-and-add instructions, which is currently not added by standard compilers. We believe that the compilation from our model without this dependency is sound as well, and leave the formal proof to a future work (see also §7.5).

the DRF-RA guarantee as illustrated below:

$$
\begin{array}{l|l}
a := \texttt{FADD}^{\texttt{acqrel}}(X,1)\ \ \textit{// 0} & b := \texttt{FADD}^{\texttt{acqrel}}(X,1)\ \ \textit{// 0} \\
\texttt{if}\ a = 0\ \texttt{then} & \texttt{if}\ b = 0\ \texttt{then} \\
\quad Y^{\texttt{rlx}} := 1 & \quad c := Y^{\texttt{rlx}}\ \ \textit{// 1} \\
 & \quad \texttt{if}\ c = 1\ \texttt{then} \\
 & \quad\quad X^{\texttt{rlx}} := 0
\end{array}
\tag{CDRF}
$$

Under RA semantics only one thread can enter the if-branch, and the only race is between the two FADDs. Hence, to maintain DRF-RA, we need to disallow the annotated behavior where both threads read 0 from X. To prevent this behavior, we need to disallow thread 1 to promise $Y := 1$ in the beginning of the run. Indeed, by reading such a promise, thread 2 can write $Y := 0$, and then, thread 1 can succeed its RMW to X and fulfill its outstanding promise. However, if we completely ignore the possible interference by other threads, thread 1 may promise $Y := 1$, as it can be certified in a local run of thread 1 that starts from the initial memory and reads the initial message of X.

Abstractly, what went wrong is that two threads compete on the same resource (*i.e.*, to perform an RMW reading from the initialization message); one of them makes a promise assuming it will get the resource first but the other thread wins the competition in the actual run. This not only causes deadlock (which is semantically inconsequential), but also breaks DRF-RA.

To address this, PS followed a simple approach: it required that threads certify their promises starting from *any* extension of the current memory. One such particular extension is the memory that will arise when the required resource is acquired by some other thread. Hence, this condition does not allow threads to promise writes assuming they will win a competition on some resource.

Revisiting CDRF, PS's certification condition blocks the promise of $Y := 1$. For example, when certifying against M_{future} that, in addition to the initialization messages, consists of a message $m = \langle X@(0, \_], 42, \_\rangle$, thread 1 is forced to read from m when performing its FADD, and cannot fulfill its promise. Since M_{future} is a possible future memory of the initial memory, thread 1 cannot promise $Y := 1$.

PS's future memory quantification maintains the thread-locality principle and

suffices for establishing DRF-RA. However, next, we demonstrate that this very conservative over-approximation of possible interference is too strong to support inter-thread optimizations, and it is also the source of unsoundness of the intended compilation scheme to Armv8.

Value-Range Analysis PS does not support inter-thread optimizations based on value-range analysis. To see this, consider a variant of the LB-G program above that does not have the redundant write to X in thread 2 and has a `CAS` instruction to X instead of the read in thread 1.

$$\begin{array}{l} a := \texttt{CAS}(X, 0, 1) \;\; /\!/ \, 1 \\ \texttt{if } a < 10 \texttt{ then} \\ \quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ \, 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{GA}$$

In PS, the annotated behavior is disallowed. Indeed, to obtain this behavior, thread 1 has to promise $Y^{\texttt{rlx}} := 1$. This promise, however, cannot be certified for every future memory M_{future}. For example, if, in addition to the initialization messages, the future memory M_{future} consists of a single message of the form $\langle X@(0, \_], 57, \_\rangle$, then the `CAS` instruction can only read 57, and the write $Y^{\texttt{rlx}} := 1$ is not executed. However, by observing the global invariant $X < 10 \wedge Y < 10$, a global compiler analysis may transform this program to the following:

$$\begin{array}{l} a := \texttt{CAS}(X, 0, 1) \;\; /\!/ \, 1 \\ Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ \, 1 \\ X^{\texttt{rlx}} := b \end{array}$$

Now, the annotated behavior is allowed (the promise $Y^{\texttt{rlx}} := 1$ is not blocked anymore), rendering the optimization unsound. This is particularly unsatisfying because PS ensures that $X < 10$ is globally valid in this program (via its "invariant logic" [33, §5.5]), but does not allow an optimizing compiler to make use of this fact.

Register Promotion A similar problem arises for a different kind of inter-thread optimization, namely *register promotion*:

$$\begin{array}{l} a := X^{\texttt{rlx}} \;\; /\!/ \, 1 \\ c := \texttt{FADD}(Z, a) \;\; /\!/ \, 0 \\ Y^{\texttt{rlx}} := 1 + c \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ \, 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{RP}$$

PS disallows the annotated behavior. Again, thread 1 cannot promise $Y^{\texttt{rlx}} := 1$, since an arbitrary future memory may not allow it to read $Z = 0$ when performing the RMW. (Note also the RMW writing $Z^{\texttt{rlx}} := 1$ cannot be promised before $Y^{\texttt{rlx}} := 1$ since it requires to read $X^{\texttt{rlx}} := 1$ first.) Nevertheless, a global compiler analysis may notice that Z is a local variable in the source program, and perform register promotion, replacing $c := \texttt{FADD}(Z, a)$ with $c := 0$ (since this FADD always returns 0). Now, PS *allows* the annotated behavior (nothing blocks the promise $Y^{\texttt{rlx}} := 1$), rendering register promotion unsound.

Unsound Compilation Scheme to Armv8 A different problem in PS, found while formally establishing the correctness of compilation to Armv8 [57], is that the intended mapping of RMWs to Armv8 is broken. In fact, this problem stems from the exact same reason as the two problems above.

While PS disallows the annotated behavior of the RP program above, when following the intended mapping to Armv8 [13], Armv8 allows the annotated behavior for the target program.[3] Roughly speaking, although the instructions cannot be reordered at the source level, they can be reordered at the micro-architecture level. FADD is effectively turned into *two* special instructions, a load exclusive followed by a store exclusive. Since there is no dependency between the load of X and the exclusive load of Z, the two loads could be executed out of order. Similarly, the two stores could be executed out of order, and so the store to Y could effectively be executed before the load of X, which in turn leads to the annotated behavior.

What went wrong? These three problems all arise because PS's certification requirement against every memory extension is overly conservative in approximating the interference by other threads. The challenge lies in relaxing this condition in a way that will ensure the soundness of inter-thread optimizations while maintaining *thread-locality*.

As CDRF shows, simply relaxing the certification requirement by requiring certification only against the current memory is not an option. Another naive remedy

[3] Here the fact that no other thread accesses Z is immaterial. Armv8 allows this behavior also when, say, a third thread executes $Z^{\texttt{rlx}} := 5$.

would be to restrict the certification to extensions of the current memory that can actually arise in the given program. This approach, however, is bound to fail:

- First, due to the intricate interaction with local optimizations, a precise approximation of other threads effect on memory is too strong—we may have a preceding local optimization that reduces the behaviors of the other threads. For instance, consider the following program:

$$\begin{array}{l|l} a := \texttt{CAS}(X, 0, 1) \;\; \textit{// 1} & X^{\texttt{rlx}} := 42 \\ \texttt{if } a < 10 \texttt{ then} & b := Y^{\texttt{rlx}} \;\; \textit{// 1} \\ \quad Y^{\texttt{rlx}} := 1 & X^{\texttt{rlx}} := b \end{array} \tag{GA+E}$$

  Here, $X^{\texttt{rlx}} := 42$ occurs in a possible future memory, but a compiler may soundly eliminate this write.

- Second, this approach is not thread-local, and, since other threads may promise as well, it immediately leads to troublesome cyclic reasoning: whether thread 1 may promise a write depends on behavior of thread 2 that may include promise steps that again depend on behavior of thread 1.

5 Solution Overview

In this section, we present the key ideas behind our modified PS model, which we call PS2. Section 5.1 describes the notion of *capped memory*, which enables value-range analysis, while §5.2 discusses *reservations*, an additional mechanism needed to support register promotion and recover the correctness of the mapping to Armv8. Section 5.3 discusses our modeling of undefined behavior (which we use to formally specify value range analysis). Finally, §5.4 describes certain trade-offs in our model.

5.1 Capped Memory

We note that PS's certification against every memory extension is quantifying over two aspects of possible interference: message *values* and message *views*.

We observe that quantifying only over message views suffices for DRF-RA. By carefully analyzing CDRF, we can see that for DRF-RA, one has to make sure that

during the certification of promises, no *acquire-release RMW* reads from a message that already exists in the memory. Indeed, (*i*) due to interference by other threads, such RMW may not have the opportunity to read from that message in the actual run; and (*ii*) such racy RMWs may exist (the DRF-RA assumption does not prevent them). Together, (i) and (ii) invalidate the DRF-RA guarantee (as happens in CDRF). We observe here that this is the *only* role of the future memory quantification that is required for ensuring DRF-RA.

The conservative future memory quantification of PS indeed disallows such problematic RMWs during certification. In fact, even certification against memory extensions that do not introduce new values in the future memory suffices for DRF-RA. For example, in CDRF, when certifying against M_{future} that, in addition to the initialization messages, has a message form $m = \langle X@(0, \_], 0, V_{\mathtt{m}} \rangle$ with $V_{\mathtt{m}}(Y) \geq t$, thread 1 is forced to read m when performing its `FADD`. Since it is an acquire `FADD`, it will increase the thread view of Y to $V_{\mathtt{m}}(Y)$, which will not allow it to fulfill its promise. More generally, when a thread promises a message of the form $\langle X@(f, t], v, V \rangle$ in the current memory M, there is always a possible memory extension M_{future} of M that forces (non-promised) RMWs of location Y performed during certification (which read from a message in M_{future}) to read from a specific message $m^{Y}_{\text{future}} \in M_{\text{future}}$ whose view of X is greater than or equal to t. When such RMWs are *acquire* RMWs, this will force the thread to increase its view of X to at least t, which, in turn, does not allow the thread to fulfill its promise.

Remark 1. Completely disallowing release-acquire RMWs during certification is too strong. We should allow them to read from local writes added during certification, since no other thread can prevent them from doing so.

We further observe that value-range analysis concerns message *values*, but it is *insensitive* to message *views*. As we saw for the GA program above, the conservative future memory quantification of PS is doing too much: it forbids any promise that depends on the value read by an RMW, which invalidates value-range analysis. However, we note that there is no problem in disallowing the following variant of GA that

uses an acquire CAS instead of a relaxed one:

$$\begin{array}{l} a := \texttt{CAS}^{\texttt{acq}}(X, 0, 1) \;\; /\!/ 1 \\ \texttt{if } a < 10 \texttt{ then} \\ \quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{GAacq}$$

Although value analysis may deduce that $a < 10$ is always true, it cannot justify the reordering of $a := \texttt{CAS}^{\texttt{acq}}(X, 0, 1)$ and $Y^{\texttt{rlx}} := 1$, since acquire accesses in general cannot be reordered with subsequent accesses. In other words, an analysis that is based solely of values does not give any information about the views of read messages, so that any optimization based on such analysis cannot enable reordering of acquire RMWs.

Based on these observations, it seems natural to replace the conservative future memory quantification of PS with a requirement to certify against all extensions of the current memory M that employ values that already exist in M (for each location). While this approach makes value-range analysis sound and maintains DRF-RA, it is still too strong for the combination of local and inter-thread optimizations. Indeed, consider the following variant of the GA+E program above.

$$\begin{array}{l} X^{\texttt{rlx}} := 42 \\ X^{\texttt{rlx}} := 0 \\ flag^{\texttt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} f := flag^{\texttt{rlx}} \\ \texttt{if } f = 1 \texttt{ then} \\ \quad a := \texttt{CAS}(X, 0, 1) \;\; /\!/ 1 \\ \quad \texttt{if } a < 10 \texttt{ then} \\ \quad\quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\ X^{\texttt{rlx}} := b \end{array} \tag{GA+E'}$$

In order for thread 2 to promise $Y^{\texttt{rlx}} := 1$, the write to $flag$ has to be executed first. (Note that a release write cannot be promised.) Therefore, the value 42 for X exists in memory when the promise $Y^{\texttt{rlx}} := 1$ is made, but, to support both the elimination of overwritten values and global value analysis, $X^{\texttt{rlx}} := 42$ should not be considered as a possible extension of the current memory. We observe that it is enough, however, to consider memory extensions whose additional messages *only* use values of maximal messages (which were not yet overwritten) to each location.

Now, instead of quantifying over a restricted set of memory extensions, we identify the most restrictive such extension, which we called the *"capped memory"*. This leads

to a conceptually simpler certification condition, where certification is needed only against one particular memory, which is uniquely determined by the current memory. The capped memory $\hat{M}$ of a memory M is obtained by:

- Filling all "gaps" between existing messages so that non-promised RMWs can only read from the maximal message of the relevant location. In other words, for every two messages $m_1 = \langle X@(\_, t], \_, \_\rangle$ and $m_2 = \langle X@(f, \_], \_, \_\rangle$ with $t < f$ and no message in between, we block the space between t and f. (The exact mechanism to achieve this, "reservations", is discussed in §5.2.)

- For every location X, attaching a "cap message" $\hat{m}_X$ with a *globally maximal view* to the latest message to X in M:

$$\hat{m}_X = \langle X@(\hat{t}_X, \hat{t}_X + 1], \hat{v}_X, \hat{V}_M\rangle$$

  where $\hat{t}_X$ and $\hat{v}_X$ are the "to"-timestamp and the value of the message to X in M with the maximal "to"-timestamp, and $\hat{V}_M$ is given by:

$$\hat{V}_M = \lambda Y.\ \max\{t \mid \langle Y@(\_, t], \_, \_\rangle \in M\}.$$

Fig. III.1 depicts an example of the capped memory construction. The shaded area in $\hat{M}$ represents the blocked space.

Starting from $\hat{M}$, any (non-promised) RMWs reading from a message in $\hat{M}$ are forced to read from the $\hat{m}_X$ messages (since the timestamp interval $[0, \hat{t}_X]$ is completely occupied). Because these messages carry maximal views, acquire RMWs reading from them cannot be executed during certification, as it will increase the thread view to $\hat{V}_M$, which, in turn, will prevent the thread from fulfilling its outstanding promises.

In turn, the new machine step is then simplified as follows:

$$\frac{\langle \mathcal{T}(\tau), M\rangle \to^+ \langle T', M'\rangle \qquad \exists T''.\ \langle T', \hat{M}'\rangle \to^* \langle T'', \_\rangle \wedge T''.\mathtt{prm} = \emptyset}{\langle \mathcal{T}, M\rangle \to \langle \mathcal{T}[\tau \mapsto T'], M'\rangle}$$

Since the capped memory is clearly one possible future memory, the semantics we obtain is clearly weaker than PS. It is (*i*) weak enough to allow the annotated behaviors of GA and RP above: certification against the capped memory will not lead to $a \geq 10$ in GA and to $c \neq 0$ in RP; and, on the other hand, (*ii*) strong enough to forbid

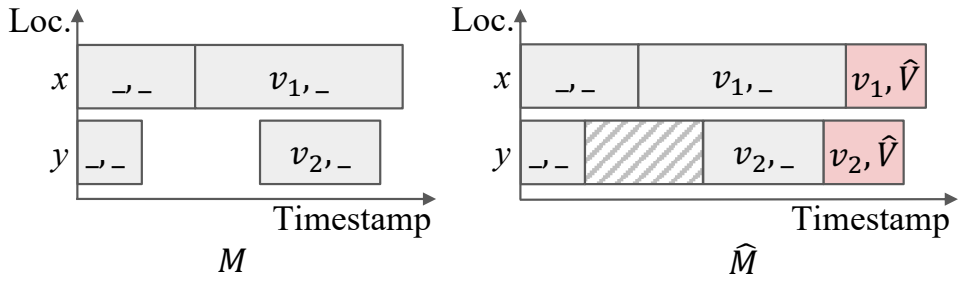


Figure III.1: An example of the capped memory

the annotated behavior of CDRF above: certification against the capped memory will not allow the $Y^{\mathtt{rlx}} := 1$ promise. In particular, by using the maximal messages for constructing capped memory, thread 2 of GA+E' can promise $Y^{\mathtt{rlx}} := 1$ and certify it while the message $X^{\mathtt{rlx}} := 42$ (which is overwritten by $X^{\mathtt{rlx}} := 0$) is in the memory.

Remark 2. The original PS quantification over all future memories could equivalently quantify over all memories defined just like the capped memory, except for using arbitrary values for the cap messages. Capped memory is more than that: it sets the value of each cap messages to that of the corresponding maximal message.

5.2 Reservations

While capped memory suffices for justifying the weak outcomes of the examples seen so far, it is still too strong to support register promotion and to validate the intended mapping to Armv8. Consider the following variant of RP that uses an *acquire* RMW in thread 1.

$$\begin{array}{l} a := X^{\mathtt{rlx}} \;\; /\!/ \, 1 \\ c := \mathtt{FADD}^{\mathtt{acq}}(Z, a) \;\; /\!/ \, 0 \\ Y^{\mathtt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := Y^{\mathtt{rlx}} \;\; /\!/ \, 1 \\ X^{\mathtt{rlx}} := b \end{array} \qquad \text{(RPacq)}$$

The weakening of PS presented in §5.1 disallows the annotated behavior. Thread 1 cannot promise $Y^{\mathtt{rlx}} := 1$ because its certification has to execute a non-promised *acquire* RMW reading from an existing message against the capped memory; and also it cannot promise the RMW $Z^{\mathtt{rlx}} := 1$ before $Y^{\mathtt{rlx}} := 1$ because its certification requires reading $X^{\mathtt{rlx}} := 1$. Nevertheless, as for RP, a global analysis may notice that Z is accessed only by one thread and perform register promotion, yielding the annotated

outcome. (Similarly, Armv8 allows the annotated behavior of the corresponding target program.)

We note that the standard (Java) optimization of removing locks used by only one thread requires to perform register promotion on local locations accessed by acquire RMWs. Indeed, lock acquisitions are essentially acquire RMWs.

So, how can we allow such behaviors without harming DRF-RA? Our idea here is to enhance PS by allowing one to declare which thread will win the competition to perform an RMW reading from a given message m. Once such a declaration is made, RMWs performed by other threads cannot read from m.

The technical mechanism for these declarations is simple: we add a "reservation" step to PS, allowing a thread to reserve a timestamp interval that it plans to use later, without committing on how it will use it (what value and view will be picked). Once an interval is reserved, other threads are blocked from reusing timestamps in this interval. Intuitively, a reservation corresponds to promising the "read part" of the RMW, which confines the behavior of other threads. In particular, if a thread reserves an interval $(t_1, t_2]$ attached to some message $(f, t_1]$, then other threads cannot read from the $(f, t_1]$ message with an RMW operation.

Since reservations are included in the machine memory (just like normal writes and promises), the semantics remains thread-local. Technically, reservations take the form $\langle X@(f, t]\rangle$ where $X \in \mathsf{Loc}$ and $(f, t]$ is a timestamp interval. To meet their purpose, we allow attaching reservations only immediately after existing concrete messages (f should be the "to"-timestamp of some existing message to the same location). Threads are also allowed to cancel their reservations (provided they can still certify their outstanding promises) if they no longer need to block an interval.

Remark 3. For the soundness of register promotion, a thread should be allowed to cancel its reservations instead of fulfilling them. Consider the following variant of the

RPacq program:

$$
\begin{array}{l}
a := X^{\texttt{rlx}} \;\; /\!/ 1 \\
\texttt{if } a = 1 \texttt{ then} \\
\quad c := \texttt{FADD}^{\texttt{acq}}(Z, 1) \\
\texttt{else} \\
\quad d := \texttt{FADD}^{\texttt{acq}}(W, 1) \\
Y^{\texttt{rlx}} := 1
\end{array}
\;\Big\|\;
\begin{array}{l}
b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\
X^{\texttt{rlx}} := b
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
a := X^{\texttt{rlx}} \;\; /\!/ 1 \\
Y^{\texttt{rlx}} := 1
\end{array}
\;\Big\|\;
\begin{array}{l}
b := Y^{\texttt{rlx}} \;\; /\!/ 1 \\
X^{\texttt{rlx}} := b
\end{array}
$$

Here, a compiler may transform the program on the left to the program on the right by promoting locations Z and W. However, if a reservation should always be fulfilled with a concrete message eventually, this optimization is unsound because the annotated behavior $a = b = 1$ can be introduced by the optimization. Indeed, to observe this behavior before the optimization, thread 1 should be able to promise $Y^{\texttt{rlx}} := 1$ before the read from X. To certify this promise, thread 1 should make a reservation to W because it has to perform an RMW to W during the certification. Then, once the thread makes a reservation to W for certifying the promise $Y = 1$, it can never be able to read 1 from X because the thread is enforced to enter the else-branch and fulfill the reservation to W. Therefore, PS2 allows a thread to cancel its reservations instead of fulfilling them.

Returning to the RPacq program above, reservations allow the annotated outcome. Thread 1 can first reserve the interval $(0, 1]$ for Z. Then, it can promise $Y^{\texttt{rlx}} := 1$ and certify its promise by using its own reservation to perform the RMW.

Intuitively, reservations are closer to the implementation of RMWs in Arm: reserving the read part of an RMW first and then writing the RMW at the reserved space later corresponds to execution of a load exclusive first and a (successful) write exclusive later.

Reservations are also used for defining the capped memory to fill the gaps between messages to the same location (§5.1). In the presence of reservations, however, the capped memory definition requires some care. First, the value of the cap messages $\hat{m}_X$ should be the value of the maximal *concrete* message to x (reservations do not carry values). Second, when constructing the capped memory for thread τ, if the maximal message to some location Y is a reservation of thread τ itself, then we do not add a

cap message for Y. In effect, during certification, the thread can execute any RMW on Y but only after filling the reserved space on Y. Other threads cannot execute an RMW on reservations of thread τ, and so cannot interfere with respect to Y.

5.3 Undefined Behavior

So far, we have described value-range optimizations by informally referring to a global analysis performed by the compiler. For our formal development, we introduce *undefined behavior* (UB). We note that UB, which is not supported in the original PS model, is also useful in a broader context (*e.g.*, to give sensible semantics to expressions like $X/0$).

In order to formally define inter-thread optimizations, we include in our language an abort instruction, `abort`, which causes UB. In turn, for a global invariant I (formally defined in §7.2), we allow the program transformation introducing at arbitrary program points the instruction `assert`(I), which is a syntactic sugar to `if` $\neg I$ `then abort`. This paves the way to further *local* optimizations, such as:

$$\begin{array}{l} \texttt{assert}(X \in \{0,1\}) \\ a := X^{\texttt{rlx}} \\ \texttt{if } a \in \{0,1\} \texttt{ then } c \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := X^{\texttt{rlx}} \\ c \end{array}$$

The standard semantics of UB is "catch-fire": UB should be thought as allowing any arbitrary sequence of operations. This enables common compiler optimizations (*e.g.*, `if` e `then` c `else abort` $\rightsquigarrow c$). Nevertheless, to make sure the semantics is not overly weak, like any thread step, for taking an `abort`-step, the certification condition has to be satisfied (where the certifying thread may replace `abort` by any sequence of operations).

Our formal condition for taking an `abort`-step is somewhat simpler: we require that for every location X, the current view of the aborting thread for X should be lower than the "to"-timestamp of all the outstanding promises for X of that thread. We say a thread is *promise-consistent* when this condition is met. Recall that a thread can take a write step to a location X when the thread view of X is lower than the "to"-timestamp of the writing message. In turn, considering that taking an `abort`-step is capable of executing arbitrary write instructions, a thread is able to fulfill its outstanding promises when aborting if and only if it is promise-consistent.

5.4 Relaxed RMWs in Certifications

In PS2, we opted to allow relaxed RMWs (that were non-promised before and read from a message that exists in the current memory) during certification of promises. This design choice can cause execution deadlocks:

$$\begin{array}{l} a := \texttt{FADD}(X, 1) \;\; \textit{// 0} \\ Y^{\texttt{rlx}} := 1 + a \end{array} \;\Big\|\; b := \texttt{FADD}(X, 1) \qquad \text{(deadlock)}$$

Suppose that the thread 1 promises $Y^{\texttt{rlx}} := 1$. This promise can be certified against the capped memory by updating the cap message of X (whose value is 0). Now, thread 2 can perform its RMW, and block thread 1 from fulfilling its promise. Although allowing such deadlocks is awkward, they are inconsequential, since deadlocking runs are discarded from the definition of observable behavior.

Similarly, this choice enables somewhat dubious behaviors that seem to invalidate atomicity of relaxed RMWs: for instance, CDRF can have the annotated behavior if one `FADD` is made `rlx`. Such behaviors are unavoidable if all (intra- and inter-thread) optimizations allowed by PS2 are supported. To see this, consider the following variant of CDRF:

$$\begin{array}{l} a := \texttt{CAS}(X, 0, 1) \\ \texttt{if } a \leq 1 \texttt{ then} \\ \quad Y^{\texttt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := \texttt{WCAS}(X, 0, 2) \\ \texttt{if } b \texttt{ then} \\ \quad c := Y^{\texttt{rlx}} \;\; \textit{// 1} \\ \quad \texttt{if } c = 1 \texttt{ then} \\ \quad\quad X^{\texttt{rlx}} := 0 \end{array} \qquad \text{(CDRF-weak)}$$

Here, a *weak compare-and-swap* operation `WCAS` is allowed to spuriously fail even if it reads the exact value it expects to update.[4] We assume that `WCAS` returns a boolean flag representing whether the update was successful. PS2 allows this program to exhibit the annotated behavior $c = 1$, which requires both threads to succeed the updates to X. This might seem to be an overly weak behavior: when thread 2 succeeds `WCAS` (and updates X to 2), it cannot read 1 from Y since thread 1 cannot update X from 0 to 1.

However, this behavior is allowed after applying a sequence of intra- and inter-thread optimizations. First, as shown in Fig. III.2, the program can be transformed into

[4] A weak compare-and-swap operation is analogous to `compare_exchange_weak` of C/C++.

(1)	(2)	(3)	(4)

$c := Y^{\texttt{rlx}}$
$b := \texttt{WCAS}(X, 0, 2)$
`if` $c = 1$ `then`
  `if` b `then`
  $X^{\texttt{rlx}} := 0$
	$c := Y^{\texttt{rlx}}$
	`if` $c = 1$ `then`
	  $b := \texttt{WCAS}(X, 0, 2)$
	  `if` b `then`
	  $X^{\texttt{rlx}} := 0$
	`else`
	  $\_ := \texttt{WCAS}(X, 0, 2)$
		$c := Y^{\texttt{rlx}}$
		`if` $c = 1$ `then`
		  $b := \texttt{WCAS}(X, 0, 2)$
		  `if` b `then`
		  $X^{\texttt{rlx}} := 0$
		`else`
		  $\_ := X^{\texttt{rlx}}$
			$c := Y^{\texttt{rlx}}$
			`if` $c = 1$ `then`
			  $b := \texttt{WCAS}(X, 0, 0)$
			`else`
			  $\_ := X^{\texttt{rlx}}$

(1) reorder an RMW $b := \texttt{WCAS}(X, 0, 0)$ follwed by a read $c := Y^{\texttt{rlx}}$ by introducing a relaxed read in the else-branch;

(2) distribute the RMW to X into both branches;

(3) replace the RMW $b := \texttt{WCAS}(X, 0, 0)$ in the else-branch with a relaxed read since a weak compare-and-swap operation may always fail; and

(4) merge the RMW $b := \texttt{WCAS}(X, 0, 0)$ with a write $X^{\texttt{rlx}} := 0$, which is executed only when the RMW succeeds.

Figure III.2: A sequence of thread-local optimizations applied to the second thread of CDRF-weak program.

the following by applying local optimizations to thread 2[5]:

$a := \texttt{CAS}(X, 0, 1)$
`if` $a \leq 1$ `then`
  $Y^{\texttt{rlx}} := 1$
	$c := Y^{\texttt{rlx}}$
	`if` $c = 1$ `then`
	  $b := \texttt{WCAS}(X, 0, 0)$
	`else`
	  $\_ := X^{\texttt{rlx}}$

Here, a global invariant $X \leq 1 \wedge Y \leq 1$ holds, and a branch condition $a \leq 1$ of thread 1 can be optimized away. Then, the RMW $b := \texttt{WCAS}(X, 0, 0)$ and the write $Y^{\texttt{rlx}} := 1$

[5]The third optimization that replaces a weak compare-and-swap operation with a read may not be sound in general particularly under a liveness-aware semantics. Cho et al. [17] provides a more involved version of this example that does not include this questionable optimization (see LDRF-PF-Fail example).

can be reordered, and the behavior $c = 1$ is allowed by a simple interleaving execution.

A stronger alternative would be to disallow relaxed RMWs during certification unless they were promised before the certification, or they read from a message that is added to the memory during certification. This can be easily achieved by defining the capped memory (against which threads certify their promises) to include a reservation instead of a cap message, which disallows to read from cap messages during certification. The resulting model is deadlock-free and it supports all (intra- and inter-thread) optimizations supported by PS2, except for the local reordering of a relaxed RMW followed by a write. To see this consider the following example:

$$\begin{array}{l|l} a := \texttt{FADD}(X, 1) \;\; /\!/ \, 1 & b := Y^{\texttt{rlx}} \;\; /\!/ \, 1 \\ Y^{\texttt{rlx}} := 1 & X^{\texttt{rlx}} := b \end{array} \qquad \text{(LB-RMW)}$$

To read the annotated values, the run must start with thread 1 promising $Y^{\texttt{rlx}} := 1$. Such a promise can only be certified if we allow relaxed RMWs that read an existing message during certification. Nevertheless, reordering the two instructions in thread 1 clearly exhibits the annotated behavior. In particular, since ARMv8 performs such reorderings, the mapping to ARMv8 should always include a dependency from relaxed RMWs, thereby incurring some (probably small) overhead.

6 Formal Model

In this section, we present our formal model, called PS2, which combines and makes precise the ideas outlined above. Here, we consider the full model including fences, release sequences, and mechanisms for modifying existing promises (split and lower). We refer the readers to [33] for the details of these additional features since they are handled just like in PS.

To keep the presentation simple and abstract, we do not fix a particular programming language syntax, and rather assume that the thread semantics is already provided as a labeled transition system, with transition labels `silent` for a silent thread transition with no memory effect, $\texttt{R}(o_{\texttt{R}}, X, v)$ for reads, $\texttt{W}(o_{\texttt{W}}, X, v)$ for writes, $\texttt{RMW}(o_{\texttt{R}}, o_{\texttt{W}}, X, v_{\texttt{R}}, v_{\texttt{W}})$ for RMWs, $\texttt{F}(o_{\texttt{F}})$ for fences, `fail` for failing assertions, and $\texttt{sys}(v)$ for a system calls.

$v \in \mathsf{Val}$	value	$S \in \mathsf{View}$	sc view
$X, Y, Z \in \mathsf{Loc}$	location	$m = \langle X@(f,t], v, V\rangle \in \mathsf{Msg}$	message
$o_\mathsf{R} \in \{\mathtt{rlx}, \mathtt{acq}\}$	read access mode	$r = \langle X@(f,t]\rangle \in \mathsf{Rsv}$	reservation
$o_\mathsf{W} \in \{\mathtt{rlx}, \mathtt{rel}\}$	write access mode	$P \subseteq \mathsf{Msg} \cup \mathsf{Rsv}$	promise set
$o_\mathsf{F} \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$	fence access mode	$M \subseteq \mathsf{Msg} \cup \mathsf{Rsv}$	memory
$\tau \in \mathsf{Tid} \triangleq \{\tau_1, \tau_2, ...\}$	thread identifier	σ	program state
$f, t \in \mathsf{Time} \triangleq \{0\} \cup \mathbb{Q}^+$	timestamp	$\mathcal{V} = \langle V_\mathtt{rel}, V_\mathtt{cur}, V_\mathtt{acq}\rangle$	thread view
$V \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$	view	$T = \langle \sigma, \mathcal{V}, P\rangle \in \mathsf{Lts}$	thread state
$V_\mathtt{rel} \in \mathsf{Loc} \to \mathsf{View}$	release view	$\langle T, S, M\rangle$	thread configuration
$V_\mathtt{cur} \in \mathsf{View}$	current view	$\mathcal{T} \in \mathsf{Tid} \to \mathsf{Lts}$	thread state mapping
$V_\mathtt{acq} \in \mathsf{View}$	acquire view	$\langle \mathcal{T}, S, M\rangle$	machine state

(MEMORY: NEW)

$$\frac{}{\langle P, M\rangle \xrightarrow{m} \langle P, M \overset{\triangle}{\hookleftarrow} m\rangle}$$

(MEMORY: FULFILL)

$$\frac{\hookleftarrow \in \{\overset{S}{\hookleftarrow}, \overset{\perp}{\hookleftarrow}\} \qquad P' = P \hookleftarrow m \qquad M' = M \hookleftarrow m}{\langle P, M\rangle \xrightarrow{m} \langle P' \setminus \{m\}, M'\rangle}$$

(READ-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(\_, t], \_, V_\mathtt{m}\rangle \\ V_\mathtt{cur}(X) \le t \qquad V_s = [X \mapsto t] \\ V'_\mathtt{cur} = V_\mathtt{cur} \sqcup V_s \sqcup (o_\mathsf{R} \sqsupseteq \mathtt{acq}\,?\,V_\mathtt{m}) \\ V'_\mathtt{acq} = V_\mathtt{acq} \sqcup V_s \sqcup (o_\mathsf{R} \sqsupseteq \mathtt{rlx}\,?\,V_\mathtt{m}) \end{array}}{\langle V_\mathtt{rel}, V_\mathtt{cur}, V_\mathtt{acq}\rangle \xrightarrow{o_\mathsf{R}, m}_\mathsf{R} \langle V_\mathtt{rel}, V'_\mathtt{cur}, V'_\mathtt{acq}\rangle}$$

(WRITE-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(f,t], \_, V_\mathtt{m}\rangle \qquad f < t \\ V_\mathtt{cur}(X) < t \qquad V_s = [X \mapsto t] \\ V'_\mathtt{cur} = V_\mathtt{cur} \sqcup V_s \qquad V'_\mathtt{acq} = V_\mathtt{acq} \sqcup V_s \\ V'_\mathtt{rel} = V_\mathtt{rel}[X \mapsto V_\mathtt{rel}(X) \sqcup V_s \sqcup (o_\mathsf{W} \sqsupseteq \mathtt{rel}\,?\,V'_\mathtt{cur})] \\ V_\mathtt{m} = (o_\mathsf{W} \sqsupseteq \mathtt{rlx}\,?\,(V'_\mathtt{rel}(X) \sqcup V_r)) \end{array}}{\langle V_\mathtt{rel}, V_\mathtt{cur}, V_\mathtt{acq}\rangle \xrightarrow{o_\mathsf{W}, V_r, m}_\mathsf{W} \langle V'_\mathtt{rel}, V'_\mathtt{cur}, V'_\mathtt{acq}\rangle}$$

Figure III.3: Formal model (domains and auxiliary definitions for the PS2 model).

The variable $o_\mathsf{R} \in \{\mathtt{rlx}, \mathtt{acq}\}$ denotes a read access mode, which is naturally ordered by $\mathtt{rlx} \sqsubset \mathtt{acq}$. Similarly, $o_\mathsf{W} \in \{\mathtt{rlx}, \mathtt{rel}\}$ denotes a write accesse mode ordered by $\mathtt{rlx} \sqsubset \mathtt{rel}$, and $o_\mathsf{F} \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$ denotes a fence ordering. An RMW label includes two access modes, a read mode and a write mode. These naturally encode the syntax of the examples we discussed above, *e.g.*,

$$\mathtt{FADD} \to \mathsf{RMW}(\mathtt{rlx}, \mathtt{rlx}, ...) \qquad \mathtt{FADD}^{\mathtt{acq}} \to \mathsf{RMW}(\mathtt{acq}, \mathtt{rlx}, ...)$$
$$\mathtt{FADD}^{\mathtt{acqrel}} \to \mathsf{RMW}(\mathtt{acq}, \mathtt{rel}, ...) \qquad \mathtt{FADD}^{\mathtt{rel}} \to \mathsf{RMW}(\mathtt{rlx}, \mathtt{rel}, ...)$$

Next, we present the components of the PS2 model.

(PROMISE)

$$\frac{m.\texttt{view} \in M' \qquad \kappa \in \{\text{A}, \text{S}, \text{L}, \text{C}\} \qquad P' = P \overset{\kappa}{\leftarrow}_p m \qquad M' = M \overset{\kappa}{\hookleftarrow} m}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \to \langle\langle \sigma, \mathcal{V}, P'\rangle, S, M'\rangle}$$

(READ)

$$\frac{\sigma \xrightarrow{\mathsf{R}(o_\mathsf{R}, X, v)} \sigma' \qquad m = \langle X@(\_, t], v, \_\rangle \in M \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m}_\mathsf{R} \mathcal{V}'}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P\rangle, S, M\rangle}$$

(WRITE)

$$\frac{\sigma \xrightarrow{\mathsf{W}(o_\mathsf{W}, X, v)} \sigma' \qquad o_\mathsf{W} = \texttt{rel} \Rightarrow \forall m' \in P(X).\ m'.\texttt{view} = \bot \qquad m = \langle X@(\_, t], v, V_\mathtt{m}\rangle \qquad \mathcal{V} \xrightarrow{o_\mathsf{W}, V_r, m}_\mathsf{W} \mathcal{V}' \qquad \langle P, M\rangle \xrightarrow{m} \langle P', M'\rangle}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, S, M'\rangle}$$

(UPDATE)

$$\frac{\sigma \xrightarrow{\mathsf{RMW}(o_\mathsf{R}, o_\mathsf{W}, X, v_r, v_w)} \sigma' \qquad o_\mathsf{W} = \texttt{rel} \Rightarrow \forall m' \in P(X).\ m'.\texttt{view} = \bot \qquad m_r = \langle X@(\_, t_r], v_r, V_r\rangle \in M \qquad m_w = \langle X@(t_r, t_w], v_w, V_w\rangle \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m_r}_\mathsf{R} \xrightarrow{o_\mathsf{W}, V_r, m_w}_\mathsf{W} \mathcal{V}' \qquad \langle P, M\rangle \xrightarrow{m_\mathsf{W}} \langle P', M'\rangle}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, S, M'\rangle}$$

(ACQ-FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F}(\texttt{acq})} \sigma' \qquad V'_\texttt{cur} = V_\texttt{acq}}{\langle\langle \sigma, \langle V_\texttt{rel}, V_\texttt{cur}, V_\texttt{acq}\rangle, P\rangle, S, M\rangle \to \langle\langle \sigma', \langle V_\texttt{rel}, V'_\texttt{cur}, V_\texttt{acq}\rangle, P\rangle, S, M\rangle}$$

(REL-FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F}(\texttt{rel})} \sigma' \qquad V'_\texttt{rel} = \lambda\_.V_\texttt{cur} \qquad \forall m \in P.\ m.\texttt{view} = \bot}{\langle\langle \sigma, \langle V_\texttt{rel}, V_\texttt{cur}, V_\texttt{acq}\rangle, P\rangle, S, M\rangle \to \langle\langle \sigma', \langle V'_\texttt{rel}, V_\texttt{cur}, V_\texttt{acq}\rangle, P\rangle, S, M\rangle}$$

(SC-FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F}(\texttt{sc})} \sigma' \qquad S' = V_\texttt{acq} \sqcup S \qquad \forall m \in P.\ m.\texttt{view} = \bot}{\langle\langle \sigma, \langle \_, \_, V_\texttt{acq}\rangle, P\rangle, S, M\rangle \to \langle\langle \sigma', \langle \lambda\_.S', S', S'\rangle, P\rangle, S', M\rangle}$$

(SYSTEM CALL)

$$\frac{\sigma \xrightarrow{\mathsf{sys}(v)} \sigma' \qquad S' = V_\texttt{acq} \sqcup S \qquad \forall m \in P.\ m.\texttt{view} = \bot}{\langle\langle \sigma, \langle \_, \_, V_\texttt{acq}\rangle, P\rangle, S, M\rangle \xrightarrow{\mathsf{sys}(v)} \langle\langle \sigma', \langle \lambda\_.S', S', S'\rangle, P\rangle, S', M\rangle}$$

(SILENT)

$$\frac{\sigma \xrightarrow{\texttt{silent}} \sigma'}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}, P\rangle, S, M\rangle}$$

(FAILURE)

$$\frac{\sigma \xrightarrow{\texttt{fail}} \sigma' \qquad \forall m \in P.\ \mathcal{V}.\texttt{cur}(m.\texttt{loc}) \le m.\texttt{to}}{\langle\langle \sigma, \mathcal{V}, P\rangle, S, M\rangle \xrightarrow{\texttt{fail}} \langle\langle \bot, \mathcal{V}, \emptyset\rangle, S, M\rangle}$$

Figure III.4: Formal model (thread transitions).

Time Time is a set of *timestamps* that is totally and densely ordered by $<$ with a minimum value, denoted by 0.

Views A *view* is a function $V : \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$. We use $\bot$ and $\sqcup$ to denote the natural bottom elements and join operations for views (pointwise extensions of the timestamp 0 and max operation on timestamps).

Concrete Messages A *concrete message* takes the form $m = \langle X@(f, t], v, V_{\mathtt{m}} \rangle$ where $X \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $f, t \in \mathsf{Time}$, and $V_{\mathtt{m}} \in \mathsf{View}$, such that $f < t$ or $f = t = 0$, and $V_{\mathtt{m}}(X) \leq t$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, $m.\mathtt{from}$, $m.\mathtt{to}$, and $m.\mathtt{view}$ the components of m.

Reservations A *reservation* takes the form $m = \langle X@(f, t] \rangle$, where $X \in \mathsf{Loc}$, and $f, t \in \mathsf{Time}$ such that $f < t$. We denote by $m.\mathtt{loc}$, $m.\mathtt{from}$, and $m.\mathtt{to}$ the components of m.

Messages A *message* is either a concrete message or a reservation. Two messages m_1 and m_2 are *disjoint*, denoted by $m_1 \# m_2$, if they have different locations or disjoint timestamp intervals:

$$\begin{aligned} m_1 \# m_2 \triangleq\ & m_1.\mathtt{loc} \neq m_2.\mathtt{loc} \ \lor \\ & m_1.\mathtt{to} < m_2.\mathtt{from} \ \lor \ m_2.\mathtt{to} < m_1.\mathtt{from} \end{aligned}$$

Two sets M_1 and M_2 of messages are disjoint, denoted by $M_1 \# M_2$, if $m_1 \# m_2$ for every $m_1 \in M_1$ and $m_2 \in M_2$.

Memory A *memory* is a (nonempty) pairwise disjoint finite set of messages. We write $M(X)$ for the sub-memory $\{ m \in M \mid m.\mathtt{loc} = X \}$ and $|M|$ for the set $\{ m \in M \mid m = \langle \_@(\_, \_], \_, \_ \rangle \}$ of concrete messages in M.

Memory Operations A memory M supports the following operations for a message m, where $m.\mathtt{loc} = X$, $m.\mathtt{from} = f$, $m.\mathtt{to} = t$, and $f < t$:

- The *additive insertion*, denoted by $M \xleftarrow{\text{add}} m$, is given by $M \cup \{m\}$. It is only defined if (*i*) $\{m\} \# M$; (*ii*) if m is a concrete message, then no message $m' \in M$ has $m'.\texttt{loc} = X$ and $m'.\texttt{from} = t$; and (*iii*) if m is a reservation, then there exists $m' \in M$ with $m'.\texttt{loc} = X$ and $m'.\texttt{to} = f$.

- The *splitting insertion*, denoted by $M \xleftarrow{\text{spl}} m$, is only defined if m is a concrete message and there exists m' in M such that $m'.\texttt{loc} = X$, $m'.\texttt{from} = f$, and $m'.\texttt{to} = t'$ with $t < t'$, in which case it is given by $M \setminus \{m'\} \cup \{m, \langle X@(t, t'], v', V'_{\mathtt{m}}\rangle\}$.

- The *lowering insertion*, denoted by $M \xleftarrow{\text{low}} m$, is only defined if m is a concrete message $\langle X@(f, t], v, V_{\mathtt{m}}\rangle$ and there exists $m' \in M$ that is identical to m except for $m.\texttt{view} \leq m'.\texttt{view}$, in which case it is given by $M \setminus \{m'\} \cup \{m\}$.

- The *cancellation*, denoted by $M \xleftarrow{\text{cnc}} m$, is given by $M \setminus \{m\}$. It is only defined if m is a reservation in M.

We use $\xleftarrow{\text{add}}_p$ to denote an additive insertion into a set of promises, which does not require the last condition of the additive insertion: for a memory P and a reservation m, $P \xleftarrow{\text{add}}_p m$ is defined if $\{m\} \# M$. To simplify the presentation, we define $\xleftarrow{\text{spl}}_p$, $\xleftarrow{\text{low}}_p$, and $\xleftarrow{\text{cnc}}_p$ to be the same as $\xleftarrow{\text{spl}}$, $\xleftarrow{\text{low}}$, and $\xleftarrow{\text{cnc}}$ respectively.

Closed View Given a view V and a memory M, we write $V \in M$ if, for every $X \in \mathsf{Loc}$, we have $V(X) = m.\texttt{to}$ for some concrete message $m \in \hat{M}(X)$.

Thread Views A *thread view* is a triple $\mathcal{V} = \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle$, where $V_{\mathtt{rel}} \in \mathsf{Loc} \to \mathsf{View}$ and $V_{\mathtt{cur}}, V_{\mathtt{acq}} \in \mathsf{View}$ satisfying $V_{\mathtt{rel}}(X) \leq V_{\mathtt{cur}} \leq V_{\mathtt{acq}}$ for all $X \in \mathsf{Loc}$. We denote by $\mathcal{V}.\texttt{rel}$, $\mathcal{V}.\texttt{cur}$, and $\mathcal{V}.\texttt{acq}$ the components of $\mathcal{V}$.

Thread States A *thread state* is a triple $T = \langle \sigma, V, P\rangle$, where σ is a local state, V is a thread view, and P is a memory. We denote by $T.\texttt{st}$, $T.\texttt{view}$, and $T.\texttt{prm}$ the components of a thread state T. We denote by $T.\texttt{st}$, $T.\texttt{view}$, and $T.\texttt{prm}$ the components of a thread state T.

Thread Configuration Steps A *thread configuration* is a triple $\langle T, S, M \rangle$, where T is a thread state, S is a timemap (the global SC timemap), and M is a memory. We use $\bot$ as a thread configuration after a failure.

To avoid repetition and simplify the presentation, we use the helper rules READ-HELPER and WRITE-HELPER shown in Fig. III.3 for defining thread configuration steps. These rules employ a couple of helpful notations: (*i*) "$[X \mapsto t]$" denotes a singleton timemap assigning t to X and 0 to other locations; and (*ii*) "$cond\,?\,X$" means X if a condition *cond* holds and $\bot$ otherwise.

Now, Fig. III.4 presents the full list of thread configuration steps.

PROMISE. A thread can take a promising step (PROMISE rule) by picking one of the memory operations and applying it to both the set of outstanding promises P and the memory M.

READ. In this step a thread reads the value of a location X from a message $m \in M$ and extend its view. Following the READ-HELPER, the thread's view of location X is extended to timestamp t. When the read is an acquire read, the view is also updated by the message view R.

WRITE and **UPDATE**. The write and the update steps cover two cases: a fresh write to memory (MEMORY:NEW) and a fulfillment of an outstanding promise (MEMORY:FULFILL). The latter allows to split the promise or lower its view before its fulfillment. When a thread writes a message m with location X along with timestamp $(\_, t]$, t extends the thread's view of location X to memory M. A release write step additionally ensures that the thread has no outstanding promise on location X. Moreover, a release write attaches the updated thread view V' to the message m. The update step is similar, except that it first reads a message with a timestamp interval $(\_, t]$, and then, writes a message with an interval $(t, \_]$.

ACQ/REL/SC-FENCE. There are three rules ACQ-FENCE, REL-FENCE, and SC-FENCE for modeling fences. A transition for an acquire-release fence (represented by a transition label `F(acqrel)`) can be captured by taking the ACQ-FENCE-step followed by the REL-FENCE-step. For release and sequentially consistent fences, a thread should make sure that there is no outstanding promise before executing taking the step.

SILENT. A thread takes a SILENT-step to perform thread-local computation which updates only the local thread state.

(MACHINE: NORMAL)

$$\frac{\langle \mathcal{T}(\tau), S, M\rangle \to^* \xrightarrow{e} \langle T', S', M'\rangle \quad \langle T', \hat{S}', \hat{M}'_{T'.\mathtt{prm}}\rangle \to^* \langle\langle \_, \_, \emptyset\rangle, \_, \_\rangle}{\langle \mathcal{T}, S, M\rangle \xrightarrow{e} \langle \mathcal{T}[\tau \mapsto T'], S', M'\rangle}$$

(MACHINE: UB)

$$\frac{\langle \mathcal{T}(\tau), S, M\rangle \to^+ \langle\langle \bot, \_, \_\rangle, S', M'\rangle}{\langle \mathcal{T}, S, M\rangle \to \langle \bot, S', M'\rangle}$$

Figure III.5: Formal model (machine transitions).

SYSTEM CALL. A thread takes a SYSTEM CALL-step that emits an event with the call's input and output values. Note that both PS and PS2 model system calls as executing a sequentially consistent fence.

FAILURE. A thread configuration $\langle T, S, M\rangle$ is only allowed to fail if T is *promise-consistent*:

$$\forall m \in T.\mathtt{prm}, T.\mathtt{view.cur}(m.\mathtt{loc}) \leq m.\mathtt{to}$$

Cap View and Messages The last message of a memory M to a location X, denoted by $\overline{m}_{M,X}$, is given by:

$$\overline{m}_{M,X} \triangleq \arg\max_{m \in M(X)} m.\mathtt{to}$$

The *cap view* of a memory M, denoted by $\hat{V}_M$, is given by:

$$\hat{V}_M \triangleq \lambda X.\ \overline{m}_{|M|,X}.\mathtt{to}$$

By definition, we have $\hat{V}_M \in M$. The *cap message* of a memory M to a location X, denoted by $\hat{m}_{M,X}$, is given by:

$$\hat{m}_{M,X} = \langle X@(\overline{m}_{M,X}.\mathtt{to}, \overline{m}_{M,X}.\mathtt{to} + 1], \overline{m}_{\hat{M},X}.\mathtt{val}, \hat{V}_M\rangle$$

Capped Memory The *capped memory* of a memory M with respect to a set of promises P, denoted by $\hat{M}_P$, is an extension of M, constructed in two steps:

1. For every $m_1, m_2 \in M$ with $m_1.\mathtt{loc} = m_2.\mathtt{loc}$, $m_1.\mathtt{to} < m_2.\mathtt{to}$, and there is no message $m' \in M(m_1.\mathtt{loc})$ such that $m_1.\mathtt{to} < m'.\mathtt{to} < m_2.\mathtt{to}$, we include a reservation $\langle m_1.\mathtt{loc@}(m_1.\mathtt{to}, m_2.\mathtt{from}]\rangle$ to $\hat{M}_P$.

2. We include a cap message $\hat{m}_{M,X}$ in $\hat{M}_P$ for every location X unless $\overline{m}_{M,X}$ is a reservation in P.

Consistency A thread configuration $\langle T, S, M\rangle$ is called *consistent* if for a *capped memory* $\hat{M}_{T.\mathtt{prm}}$ of M with respect to $T.\mathtt{prm}$ and the timemap $\hat{S} = \hat{V}_{\hat{M}_{T.\mathtt{prm}}}$ of $\hat{M}_{T.\mathtt{prm}}$, there exist T', S', M' such that:

$$\langle T, \hat{S}, \hat{M}_{T.\mathtt{prm}}\rangle \to^* \langle T', S', M'\rangle \quad \wedge \quad T'.\mathtt{prm} = \emptyset$$

Machine steps A *machine state* is a pair $\mathcal{M} = \langle \mathcal{T}, M\rangle$ consisting of a function $\mathcal{T}$ assigning a thread state to every thread, and a memory M. The initial state $\mathcal{M}^0$ (for a given program) consists of the function $\mathcal{T}^0$ mapping each thread i to its initial state σ_i^0, the $\bot$ thread view (all timestamps are 0), and an empty set of promises; and the initial memory M^0 consisting of one message $\langle X@(0,0], 0, \bot\rangle$ for each location X. The two kinds of machine steps are given in Fig. III.5. At each machine step, the thread taking the step should check that it is consistent. In the MACHINE: NORMAL rule, the consistency checking is inlined as the second premise. For the MACHINE: FAILURE rule, there is no need for certification since an undefined behavior directly fulfills any outstanding promises. Here, we use $\bot$ as a machine state after a failure.

Behaviors To define what is externally observable during executions of a program *prog*, we use the system calls that *prog*'s executions perform. More precisely, every execution induces a sequence of system calls, and the set of behaviors of *prog* under PS2, denoted $[\![\mathit{prog}]\!]_{\mathrm{PS2}}$, consists of all such sequences induced by executions of *prog*. When a `fail` occurs during the execution, $[\![\mathit{prog}]\!]_{\mathrm{PS2}}$ consists of the sequence of system calls performed before the failure followed by an arbitrary sequence of system calls (reflecting an *undefined behavior*).

7 Results

We next present the results of PS2. Except for Theorems 6 to 8 (whose proofs are given in §8), all other results are fully mechanized in the Coq proof assistant.

7.1 Intra-thread Optimizations

A transformation $prog_{\text{src}} \rightsquigarrow prog_{\text{tgt}}$ is *sound* if it does not introduce behaviors under any (parallel and sequential) context:

$$\forall\, C,\ [\![C[prog_{\text{src}}]]\!]_{\text{PS2}} \supseteq [\![C[prog_{\text{tgt}}]]\!]_{\text{PS2}}.$$

PS2 allows *all* compiler transformations supported by PS. Additionally, it supports replacing `abort` by arbitrary code (more precisely, $\texttt{abort}; C_1 \rightsquigarrow C_2$). Since $\texttt{assert}(e)$ is defined as $\texttt{if}\ \neg e\ \texttt{then abort}$, the following transformations are valid:

1. $\texttt{assert}(e); C \rightsquigarrow \texttt{assert}(e); C[\texttt{true}/e]$
2. $\texttt{assert}(e) \rightsquigarrow \texttt{skip}$

Thanks to thread-locality of PS and PS2, we proved a theorem that combines and lifts the local simulation relations (almost without any reasoning on certifications) between pairs of threads S_i, T_i into a global simulation relation between the composed programs $S_1 \parallel \ldots \parallel S_n$ and $T_1 \parallel \ldots \parallel T_n$. This theorem allows us to easily prove soundness of the thread-local transformations using sequential and thread-local simulation relations. See Kang [32] and our Coq formalization for more details.

7.2 Value-Range Optimizations

First, we provide a global value-range analysis and prove its soundness in PS2. A *value-range analysis* is a tuple $A = \langle J, S_1, \ldots, S_n \rangle$, where $J \in \mathsf{Loc} \to \mathcal{P}(\mathsf{Val})$ represents a set of possible values for each location and $S_i \subseteq \mathsf{State}_i$ a set of possible local states of the underlying language (*i.e.*, excluding the thread views) for each thread i. The analysis is *sound* for a program *prog* if (i) the initial value for each location is in J and the initial state of each thread i in *prog* is in S_i; (ii) taking a step from each state in S_i necessarily leads to a state in S_i assuming that it only reads a value in J and guaranteeing that it only writes a value in J.

Now, we show that sound analysis for *prog* holds in every reachable state of *prog*.

Theorem 1 (Soundness of Value-Range Analysis)**.** For a sound value-range analysis $\langle J, S_1, \ldots, S_n \rangle$ for *prog*, if $\langle \mathcal{T}, M \rangle$ is a reachable machine state for *prog*, then $\mathcal{T}(i).\texttt{st} \in S_i$ for every thread i, and $m.\texttt{val} \in J(X)$ for every $m \in M(X)$.

$$
\begin{array}{l}
\mathsf{promote}(s, X_p, r_p) := \\
\quad \textbf{match } s \textbf{ with} \\
\quad \mid\ s_1; s_2 \Rightarrow \mathsf{promote}(s_1, X_p, r_p); \mathsf{promote}(s_2, X_p, r_p) \\
\quad \mid\ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \Rightarrow \texttt{if } e \texttt{ then } \mathsf{promote}(s_1, X_p, r_p) \texttt{ else } \mathsf{promote}(s_2, X_p, r_p) \\
\quad \mid\ \texttt{do } s_1 \texttt{ while } e \Rightarrow \texttt{do } \mathsf{promote}(s_1, X_p, r_p) \texttt{ while } e \\
\quad \mid\ r := {X_p}^o \Rightarrow r := r_p \\
\quad \mid\ X_p^o := r \Rightarrow r_p := r \\
\quad \mid\ r := \texttt{FADD}^{o_1,o_2}(X_p, v) \Rightarrow r_p := r_p + v; r := r_p \\
\quad \mid\ r := \texttt{CAS}^{o_1,o_2}(X_p, v_{old}, v_{new}) \Rightarrow \texttt{if } r_p = v_{old} \texttt{ then } r_p := v_{new}; r := 1 \texttt{ else } r := 0 \\
\quad \mid\ \_ \Rightarrow s
\end{array}
$$

Figure III.6: An algorithm for register promotion that promotes a memory location X_p to a register r_p in statements s.

Second, we prove the soundness of inter-thread optimizations based on sound value-range analysis. An optimization based on a value-range analysis $A = \langle J, S_1, \ldots, S_n \rangle$ can be seen as inserting $\texttt{assert}(e)$ at positions in thread i when e is always evaluated to `true`. For this, we define a relation, inter_thread_opt(A, $prog_{\text{src}}$, $prog_{\text{tgt}}$), which holds when $prog_{\text{tgt}}$ is obtained from $prog_{\text{src}}$ by inserting valid assertions based on A.

Theorem 2 (Soundness of Inter-thread Optimizations)**.** For a sound value-range analysis A of $prog_{\text{src}}$, and for $prog_{\text{tgt}}$ such that inter_thread_opt(A, $prog_{\text{src}}$, $prog_{\text{tgt}}$), we have $[\![prog_{\text{src}}]\!]_{\text{PS2}} \supseteq [\![prog_{\text{tgt}}]\!]_{\text{PS2}}$.

7.3 Register Promotion

We prove soundness of register promotion. As shown in Fig. III.6, we denote by $\mathsf{promote}(s, X, r)$ the statement obtained from a statement s by promoting the accesses to memory location X to accesses to register r.

Theorem 3 (Soundness of Register Promotion)**.** For a program $s_1 \| \ldots \| s_n$, if memory location X is only accessed by s_i (*i.e.*, not occurring in s_j for every $j \neq i$) and register r is fresh in s_i (*i.e.*, not occurring in s_i), we have:

$$[\![s_1 \| \ldots \| s_n]\!]_{\text{PS2}} \supseteq [\![s_1 \| \ldots \| \mathsf{promote}(s_i, X, r) \| \ldots \| s_n]\!]_{\text{PS2}}.$$

7.4 DRF Theorems

We prove four DRF theorems for PS2: DRF-Promise, DRF-RA, DRF-Lock-RA and DRF-Lock-SC. First, we need several definitions:

- *Promise-free* (PF) semantics is the strengthening of PS2 obtained by revoking the ability to make promises or reservations.
- *Release-acquire* (RA) is the strengthening of PF obtained by interpreting all memory operations as if they have `ra` access mode.
- *Sequential consistency* (SC) is the strengthening of RA obtained by forcing every read of a location X to read from the message with location X with the maximal timestamp and every write to a location X to write a message at a timestamp higher than any other X-message.

In the absence of promises, PS and PS2 coincide:

Theorem 4. PF is equivalent to the promise-free fragment of PS, and thus the same holds for RA and SC.

We say that a machine state is `rlx`-race-free, if whenever two different threads may take a non-promise step accessing the same location and at least one of them is writing, then both are `ra` accesses.

Theorem 5 (DRF-Promise)**.** If every PF-reachable machine state for *prog* is `rlx`-race-free, then $[\![prog]\!]_{\text{PF}} = [\![prog]\!]_{\text{PS2}}$.

This theorem is one of the main results of DRF theorems established for PS2 and serves as a key lemma for for proving DRF-RA theorem that will be introduced shortly. In our Coq formalization, we proved a stronger version of DRF-Promise, which is presented in [43, §E].

Theorem 6 (DRF-RA)**.** If every RA-reachable machine state for *prog* is `rlx`-race-free, then $[\![prog]\!]_{\text{RA}} = [\![prog]\!]_{\text{PS2}}$.

Thanks to Theorems 4 and 5, the proof of DRF-RA for PS2 is identical to that for PS given in [33].

Our DRF-Lock theorems given below generalize those for PS given in [33] in two aspects: our `Lock` are implemented with an *acquire* CAS rather than *acquire-release* CAS that was assumed in [33]; and our results also cover `tryLock` that may fail to acquire the lock, not just `Lock` and `Unlock`.[6]

We define `tryLock`, `Lock` and `Unlock` as follows:

$$\begin{aligned} a := \texttt{tryLock}(L) &\triangleq a := \texttt{WCAS}^{\texttt{acq}}(L, 0, 1) \\ \texttt{Lock}(L) &\triangleq \texttt{do}\ a := \texttt{tryLock}(L)\ \texttt{while}\ !a \\ \texttt{Unlock}(L) &\triangleq L^{\texttt{rel}} := 0 \end{aligned}$$

where $\texttt{WCAS}^o$ is the weak compare-and-swap operation, which can either return `true` after successfully performing $\texttt{CAS}^o$, or return `false` after reading *any* value from L with *relaxed* mode.

We prove DRF-Lock-RA and DRF-Lock-SC for programs using the three lock operations. We say such a program is *well-locked* if (1) locations are partitioned into *lock* and *non-lock* locations, (2) lock locations are accessed only by the three lock operations, and (3) `Unlock` is executed only when the thread holds the lock.

Theorem 7 (DRF-Lock-RA)**.** For a well-locked program *prog*, if every RA-reachable machine state for *prog* is `rlx`-race-free for all non-lock locations, then $[\![prog]\!]_{\text{RA}} = [\![prog]\!]_{\text{PS2}}$.

Theorem 8 (DRF-Lock-SC)**.** For a well-locked program *prog*, if every SC-reachable machine state reachable for *prog* is race-free for all non-lock locations, then $[\![prog]\!]_{\text{SC}} = [\![prog]\!]_{\text{PS2}}$.

The pen-and-paper proofs of these theorems are given in §8.

7.5 Compilation Correctness

Following Podkopaev et al. [57], we prove the correctness of mapping from PS2 to hardware models (x86-TSO, POWER, Armv7, Armv8, RISC-V) using the Intermediate Memory Model, IMM, from which intended compilation schemes to the different architectures are already proved to be correct.

[6] `Lock` is typically implemented by repeating `tryLock` until it succeeds to acquire the lock.

Theorem 9 (Correctness of Compilation to IMM)**.** Every outcome of a program *prog* under IMM is also an outcome of *prog* under PS2, *i.e.*, $[\![prog]\!]_{\text{PS2}} \supseteq [\![prog]\!]_{\text{IMM}}$.

We note that this result (which is mechanized in Coq) requires the existence of a control dependency from the read part of each RMW operation. Such dependency exists "for free" in `CAS` operations, since its write operation (a store-conditional instruction) is anyway control-dependent on the read operation (a load-link instruction). However, when compiling `FADD`s to Armv8, the compiler has to place "fake" control dependencies to meet this condition (and be able to use our theorem). We conjecture that a slightly more efficient compilation (standard) scheme of `FADD`s that does not introduce such dependencies is also sound. We leave this proof to a future work. In any case, our result is better than the one for PS by Podkopaev et al. [57] that requires an extra barrier ("ld fence") when compiling RMWs to Armv8.

Remark 4. As in Armv8, our compilation result to RISC-V uses release/acquire accesses. These accesses are not a part of RISC-V ISA, but the RISC-V memory model (RVWMO) is "*designed to be forwards-compatible with the potential addition*" of them [69, §14.1].

8 Proofs

In this section, we provide the pen-and-paper proofs of DRF-Lock theorems, Thm. 7 and Thm. 8.

We start by analyzing an invariant of well-locked programs. If a program *prog* is *well-locked* with a set of lock locations $\mathcal{L}$, the following invariant holds for every reachable memory M during the execution of *prog*:

$$
\begin{aligned}
\forall L \in \mathcal{L}.\ & \forall \langle L@(\_,\_], v, \_\rangle \in M.\ v = 0 \ \lor\ v = 1 \ \land \\
& \forall \langle L@(t,\_], 1, V^1\rangle \in M.\ \exists \langle L@(\_,t], 0, V^0\rangle \in M \ \land \\
& \forall \langle L@(\_,t^0], v^0, V^0\rangle,\ \langle L@(\_,t^1], v^1, V^1\rangle \in M.\ t^0 \sqsubseteq t^1 \ \Rightarrow\ V^0 \sqsubseteq V^1 \ \land \\
& \forall \langle L@(f,t], 0, \_\rangle \in M.\ t = \hat{t}_L \ \lor\ \exists \langle L@(t,\_], 1, \_\rangle \in M
\end{aligned}
$$

We define $\texttt{tryLock}^{o_1,o_2}$ as $a := \texttt{WCAS}^{\texttt{acq},o_1,o_2}(L,0,1)$, where $\texttt{WCAS}^{o_0,o_1,o_2}$ is a weak compare-and-swap operation that either (*i*) returns `true` after a successful `CAS` with

a read access mode o_0 and a write access mode o_1; or (*ii*) returns `false` after reading *any* value from L with o_2 access mode. Note that `tryLock` in Thm. 7 and Thm. 8 can be represented by $\texttt{tryLock}^{\texttt{rlx},\texttt{rlx}}$.

We define `nondetLock` as follows where `choose{a, b}` non-deterministically executes one of `a` or `b`:

$$\texttt{nondetLock}^{o_1,o_2}(L) \triangleq \texttt{choose}\{0, \texttt{Lock}^{o_1,o_2}(L)\}.$$

For a program *prog*, we define $prog[o_1', o_2']$ to be a program obtained by replacing every $\texttt{tryLock}^{o_1,o_2}(L)$ in *prog* with $\texttt{tryLock}^{o_1',o_2'}(L)$, and $prog^?$ be a program obtained by replacing every $\texttt{tryLock}^{o_1,o_2}(L)$ in *prog* with $\texttt{nondetLock}^{o_1,o_2}(L)$.

With above definitions, the following lemma holds.

Lemma 1 (Strengthening Lock)**.** For a well-locked program *prog*, we have:

$$[\![prog]\!]_{\text{PS2}} = [\![prog[\texttt{ra}, \texttt{ra}]]\!]_{\text{PS2}}.$$

Proof. It is enough to show that any execution of *prog* can be simulated by an execution of $prog[\texttt{ra}, \texttt{ra}]$.

First, we define *view_attached*, *wf_attached* and $wf\_attached_{th}$:

$$\begin{aligned}
&view_attached(L, t_L, V, V) \triangleq \\
&\quad V.\texttt{rlx}(L) = t_L \Rightarrow V \sqsubseteq V \wedge V.\texttt{pln}(L) = t_L \\
&wf_attached(\langle \mathcal{T}_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}} \rangle) \triangleq \\
&\quad \forall L \in \mathcal{L},\ \langle L@(_, t_L], _, V_L \rangle \in M_{\text{src}}. \\
&\qquad (\forall i.\ (\forall X.\ view_attached(L, t_L, V_L, \mathcal{T}_{\text{src}}(i).V.\texttt{rel}(X))) \wedge \\
&\qquad\quad view_attached(L, t_L, V_L, \mathcal{T}_{\text{src}}(i).V.\texttt{cur}) \wedge \\
&\qquad\quad view_attached(L, t_L, V_L, \mathcal{T}_{\text{src}}(i).V.\texttt{acq})) \wedge \\
&\qquad\quad view_attached(L, t_L, V_L, \langle \mathcal{S}_{\text{src}}, \mathcal{S}_{\text{src}} \rangle) \wedge \\
&\qquad \forall \langle _@(_, _], _, V \rangle \in M_{\text{src}}.\ view_attached(L, t_L, V_L, V) \\
&wf_attached_{th}(\langle T_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}} \rangle) \triangleq \\
&\quad \forall L \in \mathcal{L},\ \langle L@(_, t_L], _, V_L \rangle \in M_{\text{src}}. \\
&\qquad (\forall X, view_attached(L, t_L, V_L, T_{\text{src}}.V.\texttt{rel}(X))) \wedge
\end{aligned}$$

$$
\begin{aligned}
&\mathit{view\_attached}(L, t_L, V_L, T_{\text{src}}.V.\texttt{cur}) \ \wedge \\
&\mathit{view\_attached}(L, t_L, V_L, T_{\text{src}}.V.\texttt{acq}) \ \wedge \\
&\mathit{view\_attached}(L, t_L, V_L, \langle \mathcal{S}_{\text{src}}, \mathcal{S}_{\text{src}} \rangle) \ \wedge \\
&\forall \langle \_@(\_, \_], \_, V \rangle \in M_{\text{src}}.\ \mathit{view\_attached}(L, t_L, V_L, V)
\end{aligned}
$$

Remark 5. The following properties on *view_attached* hold for every message $\langle L@(\_, t_L], \_, V \rangle \in M$:

- $\mathit{view\_attached}(L, t_L, V, \bot)$
- $\mathit{view\_attached}(L, t_L, V, V_1) \ \wedge \ \mathit{view\_attached}(L, t_L, V, V_2) \ \Rightarrow$ $\mathit{view\_attached}(L, t_L, V, V_1 \sqcup V_2)$
- $[X \mapsto t_X] \neq [L \mapsto t_L] \ \Rightarrow \ \mathit{view\_attached}(L, t_L, V, [X \mapsto t_X])$

We define $\overset{\text{ST}}{\sim}$ to be a simulation relation between program states of $\mathit{prog}[\texttt{ra}, \texttt{ra}]$ and prog. Specifically, $\sigma_{\text{src}} \overset{\text{ST}}{\sim} \sigma_{\text{tgt}}$ iff σ_{src} is the same as σ_{tgt} except every `tryLock` in the statement of σ_{src} has the ordering (`ra`, `ra`).

Then we define simulation relations between memories, thread states, and machine configurations as follows:

$$
\begin{aligned}
&M_{\text{src}} \overset{\text{M}}{\sim} M_{\text{tgt}} \ \triangleq \\
&\quad (\forall \langle X@(f, t], v, V_{\text{src}} \rangle \in M_{\text{src}}. \\
&\qquad \exists V_{\text{tgt}}.\ \langle X@(f, t], v, V_{\text{tgt}} \rangle \in M_{\text{tgt}} \ \wedge \ (X \notin \mathcal{L} \ \vee \ v \neq 1 \ \Rightarrow \ V_{\text{src}} \sqsubseteq V_{\text{tgt}})) \ \wedge \\
&\quad (\forall \langle X@(f, t], v, V_{\text{tgt}} \rangle \in M_{\text{tgt}}. \\
&\qquad (\exists V_{\text{src}}.\ \langle X@(f, t], v, V_{\text{src}} \rangle \in M_{\text{src}}) \ \vee \ (X \in L \ \wedge \ v = 1 \ \wedge \ \langle X@(f, t] \rangle \in M_{\text{src}}) \\
&T_{\text{src}} \overset{\text{T}}{\sim} T_{\text{tgt}} \ \triangleq \\
&\quad T_{\text{src}}.\texttt{st} \overset{\text{ST}}{\sim} T_{\text{tgt}}.\texttt{st} \ \wedge \\
&\quad T_{\text{src}}.\texttt{view.cur} \sqsubseteq T_{\text{tgt}}.\texttt{view.cur} \ \wedge \\
&\quad T_{\text{src}}.\texttt{view.acq} \sqsubseteq T_{\text{tgt}}.\texttt{view.acq} \ \wedge \\
&\quad (\forall X \notin \mathcal{L}.\ T_{\text{src}}.\texttt{view.rel}(X) \sqsubseteq T_{\text{tgt}}.\texttt{view.rel}(X)) \ \wedge \\
&\quad T_{\text{src}}.P \overset{\text{M}}{\sim} T_{\text{tgt}}.P \ \wedge \\
&\quad \forall \langle X@(f, t], \_, V_{\text{src}} \rangle \in T_{\text{src}}.P,\ \langle X@(f, t], \_, V_{\text{tgt}} \rangle \in T_{\text{tgt}}.P.
\end{aligned}
$$

$$
\begin{aligned}
&(T_{\text{tgt}}.\texttt{view.rel}(X) \sqsubseteq V_{\text{tgt}} \Rightarrow T_{\text{src}}.\texttt{view.rel}(X) \sqsubseteq V_{\text{src}}) \ \wedge \\
&(\forall \langle X@(\_, f], \_, V'_{\text{src}}\rangle \in T_{\text{src}}.P,\ \langle X@(\_, f], \_, V'_{\text{tgt}}\rangle \in T_{\text{tgt}}.P. \\
&\quad V'_{\text{tgt}} \sqsubseteq V_{\text{tgt}} \Rightarrow V'_{\text{src}} \sqsubseteq V_{\text{src}})
\end{aligned}
$$

$$
\begin{aligned}
&\langle T_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}}\rangle \overset{\text{S}}{\sim} \langle T_{\text{tgt}}, \mathcal{S}_{\text{tgt}}, M_{\text{tgt}}\rangle \ \triangleq \\
&\quad T_{\text{src}} \overset{\text{T}}{\sim} T_{\text{tgt}} \ \wedge \ \mathcal{S}_{\text{src}} \sqsubseteq \mathcal{S}_{\text{tgt}} \ \wedge \ M_{\text{src}} \overset{\text{T}}{\sim} M_{\text{tgt}} \ \wedge \ wf\_attached_{th}(\langle T_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}}\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\langle \mathcal{T}_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}}\rangle \overset{\text{C}}{\sim} \langle \mathcal{T}_{\text{tgt}}, \mathcal{S}_{\text{tgt}}, M_{\text{tgt}}\rangle \ \triangleq \\
&\quad (\forall i.\ \mathcal{T}_{\text{src}}(i) \overset{\text{T}}{\sim} \mathcal{T}_{\text{tgt}}(i) \ \wedge \ \mathcal{S}_{\text{src}} \sqsubseteq \mathcal{S}_{\text{tgt}} \ \wedge \ M_{\text{src}} \overset{\text{T}}{\sim} M_{\text{tgt}}) \ \wedge \\
&\quad wf\_attached(\langle \mathcal{T}_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}}\rangle)
\end{aligned}
$$

With the definition of the simulation relations, we begin by simulating *thread* steps: for any thread configurations $TC^1_{\text{src}}(= \langle T_{\text{src}}, \mathcal{S}_{\text{src}}, M_{\text{src}}\rangle), TC^1_{\text{tgt}}(= \langle T_{\text{tgt}}, \mathcal{S}_{\text{tgt}}, M_{\text{tgt}}\rangle)$, and TC^2_{tgt} such that $TC^1_{\text{src}} \overset{\text{S}}{\sim} TC^1_{\text{tgt}}$ and $TC^1_{\text{tgt}} \to TC^2_{\text{tgt}}$, there exists a thread configuration TC^2_{src} such that $TC^1_{\text{src}} \to TC^2_{\text{src}}$ and $TC^2_{\text{src}} \overset{\text{S}}{\sim} TC^2_{\text{tgt}}$. Consider the following cases of the thread step taken by the target thread configuration, $TC^1_{\text{tgt}} \to TC^2_{\text{tgt}}$:

1. `tryLock`(L) succeeds.

   T_{src} takes the same step as T_{tgt} and both $\overset{\text{T}}{\sim}$ and $\overset{\text{M}}{\sim}$ still hold. We suppose that T_{src} writes $\langle L@(\_, t], 1, V\rangle$ and the thread view is updated to $\texttt{view}'$. Then, $\texttt{view}'.\texttt{cur}.\texttt{rlx}(L) = \texttt{view}'.\texttt{cur}.\texttt{acq}(L) = t$ holds.

   Since V is a join of the source thread's relaxed view and the view of the read message, $view\_attached(L, t, V, \texttt{view}'.\texttt{cur})$ and $view\_attached(L, t, V, \texttt{view}'.\texttt{acq})$ are satisfied. Therefore, $wf\_attached_{th}$ still holds.

2. `tryLock`(L) fails.

   There exists a message $\langle L@(\_, T_{\text{src}}.\texttt{view.cur.rlx}(L)], v, V\rangle \in M_{\text{src}}$. T_{src} reads this message and *fails* to acquire the lock regardless of the value v.

   Since $view\_attached(L, T_{\text{src}}.\texttt{view.cur.rlx}(L), V, T_{\text{src}}.\texttt{view.cur.rlx})$, we can get $V^1_{\text{src}} \sqsubseteq T_{\text{src}}.\texttt{view.cur.rlx}$. Therefore, after T_{src} reads the message, the thread view of T_{src} does not increase. As the thread view of T_{src} remains the same and the thread view of T_{tgt} may increase, $\overset{\text{T}}{\sim}$ and $wf\_attached_{th}$ still hold.

3. `Unlock`(L).

   T_{src} takes the same step as T_{tgt} took and both $\overset{\text{T}}{\sim}$ and $\overset{\text{M}}{\sim}$ still hold. We suppose

that T_{src} writes $\langle L@(\_, t], 0, V\rangle$ and the thread view becomes $\texttt{view}'$.
Since V equals $T_{\text{src}}.\texttt{view.cur}$ and $T_{\text{src}}.\texttt{view.cur} \sqsubseteq \texttt{view}'.\texttt{cur} \sqsubseteq \texttt{view}'.\texttt{acq}$, $\mathit{view\_attached}(L, t, V, \texttt{view}'.\texttt{cur})$ and $\mathit{view\_attached}(L, t, V, \texttt{view}'.\texttt{acq})$ are satisfied. Therefore, $\mathit{wf\_attached}_{th}$ still holds.

4. **PROMISE** step.
Suppose that $\langle X@(f, t], v, V_{\text{tgt}}\rangle$ is the message that T_{tgt} newly promised. If $X \in \mathcal{L}$, v should be 1. Then T_{src} reserves $\langle X@(f, t]\rangle$ with the same interval. In other cases, T_{src} promises $\langle X@(f, t], v, V_{\text{src}}\rangle$, where V_{src} is determined as follows:

$$\forall L \notin \mathcal{L}.\ V_{\text{src}}.\texttt{pln}(L) = V_{\text{tgt}}.\texttt{pln}(L) \ \wedge\ V_{\text{src}}.\texttt{rlx}(L) = V_{\text{tgt}}.\texttt{rlx}(L)$$
$$\forall L \in \mathcal{L}.\ V_{\text{src}}.\texttt{pln}(L) = V_{\text{src}}.\texttt{pln}(L) = t_L$$
where t_L is a maximum timestamp such that
$$\exists \langle L@(\_, t_L], \_, V\rangle \in M_{\text{src}}.$$
$$(\forall L' \notin \mathcal{L}.\ V.\texttt{pln}(L') \sqsubseteq V_{\text{tgt}}.\texttt{pln}(L') \ \wedge\ V.\texttt{rlx}(L') \sqsubseteq V_{\text{tgt}}.\texttt{rlx}(L'))$$

By construction, $\mathit{view\_attached}(L, t_L, V_L, V_{\text{src}})$ for every $\langle L@(t_L, \_], \_, V\rangle \in M_{\text{src}}$. Therefore, $\mathit{wf\_attached}_{th}$ still holds.

5. Other steps.
T_{src} takes the same step as T_{tgt} took. Since $T_{\text{src}}.\texttt{view}$ except the release view on $L \in \mathcal{L}$ and a view of every related message in the M_{src} are lower than those of the target thread configuration, $\overset{\mathcal{M}}{\sim}$ and $\overset{\mathcal{I}}{\sim}$ still hold. The source thread's new thread view V', the new SC timemap $\mathcal{S}$, and any added messages' views are obtained by joining existing views or a singleton view that does not contain L. By Remark 5, the new source thread configuration satisfies $\mathit{wf\_attached}$.

In the same way, we can prove that the source thread can simulate the target thread's certification steps. The only difference is the *cap messages* on $X \in \mathcal{L}$. Since there is no relaxed RMW on $X \in \mathcal{L}$, capped messages does not make any changes on the above simulation arguments for thread steps. Therefore, if $T_{\text{src}} \overset{\mathcal{I}}{\sim} T_{\text{tgt}}$ and T_{tgt} is consistent, then T_{src} is consistent as well.

Now, given a machine step of the target program, we construct a machine step of

the source:

$$\begin{aligned}
&\forall\, \mathcal{M}^1_{\mathrm{src}},\ \mathcal{M}^1_{\mathrm{tgt}},\ \mathcal{M}^2_{\mathrm{tgt}}.\\
&\quad \mathcal{M}^1_{\mathrm{src}} \mathrel{\subsetneq} \mathcal{M}^1_{\mathrm{tgt}} \ \wedge\ (\mathcal{M}^1_{\mathrm{tgt}} \Rightarrow \mathcal{M}^2_{\mathrm{tgt}}) \ \Rightarrow\\
&\quad \exists\, \mathcal{M}^2_{\mathrm{src}}.\ (\mathcal{M}^1_{\mathrm{src}} \overset{l}{\Rightarrow} \mathcal{M}^2_{\mathrm{src}}) \ \wedge\ \mathcal{M}^2_{\mathrm{src}} \mathrel{\subsetneq} \mathcal{M}^2_{\mathrm{tgt}}.
\end{aligned}$$

Let's say $\mathcal{M}^1_{\mathrm{src}} = \langle \mathcal{T}^1_{\mathrm{src}}, \mathcal{S}^1_{\mathrm{src}}, M^1_{\mathrm{src}} \rangle$, $\mathcal{M}^1_{\mathrm{tgt}} = \langle \mathcal{T}^1_{\mathrm{tgt}}, \mathcal{S}^1_{\mathrm{tgt}}, M^1_{\mathrm{tgt}} \rangle$, and the i-th thread of $\mathcal{M}^2_{\mathrm{tgt}}$, $\mathcal{T}^1_{\mathrm{tgt}}(i)$ took the step, resulting in $\mathcal{M}^2_{\mathrm{tgt}} = \langle \mathcal{T}^1_{\mathrm{tgt}}[i \mapsto T^2_{\mathrm{tgt}}], \mathcal{S}^2_{\mathrm{tgt}}, M^2_{\mathrm{tgt}} \rangle$ for some T^2_{tgt}. From that $\mathcal{M}^1_{\mathrm{src}} \mathrel{\subsetneq} \mathcal{M}^1_{\mathrm{tgt}}$, we have $\mathcal{T}^1_{\mathrm{src}}(i) \mathrel{\mathrm{I}} \mathcal{T}^1_{\mathrm{tgt}}(i)$. Since the source thread configuration can simulate the target steps and indeed becomes consistent, we have the following:

$$\begin{aligned}
\exists\, T^2_{\mathrm{src}},\ \mathcal{S}^2_{\mathrm{src}},\ M^2_{\mathrm{src}}.\ (\langle \mathcal{T}^1_{\mathrm{src}}(i), \mathcal{S}^1_{\mathrm{src}}, M^1_{\mathrm{src}} \rangle \to^+ \langle T^2_{\mathrm{src}}, \mathcal{S}^2_{\mathrm{src}}, M^2_{\mathrm{src}} \rangle) \ \wedge\\
\langle T^2_{\mathrm{tgt}}, \mathcal{S}^2_{\mathrm{tgt}}, M^2_{\mathrm{tgt}} \rangle \text{ is consistent} \ \wedge\\
\langle T^2_{\mathrm{src}}, \mathcal{S}^2_{\mathrm{src}}, M^2_{\mathrm{src}} \rangle \mathrel{\mathrm{S}} \langle T^2_{\mathrm{tgt}}, \mathcal{S}^2_{\mathrm{tgt}}, M^2_{\mathrm{tgt}} \rangle.
\end{aligned}$$

while leaving the same trace as the target thread steps. Therefore, we achieve the machine step of the source machine configuration, $\mathcal{M}^1_{\mathrm{src}} \overset{l}{\Rightarrow} \langle \mathcal{T}^1_{\mathrm{src}}[i \mapsto T^2_{\mathrm{src}}], \mathcal{S}^2_{\mathrm{src}}, M^2_{\mathrm{src}} \rangle$, where $\langle \mathcal{T}^1_{\mathrm{src}}[i \mapsto T^2_{\mathrm{src}}], \mathcal{S}^2_{\mathrm{src}}, M^2_{\mathrm{src}} \rangle \mathrel{\subsetneq} \mathcal{M}^2_{\mathrm{tgt}}$.

Since the initial machine of $prog[\texttt{ra}, \texttt{ra}]$ and $prog$ are related by $\mathrel{\subsetneq}$,

$$[\![prog]\!]_{\mathrm{PS2}} \subseteq [\![prog[\texttt{ra}, \texttt{ra}]]\!]_{\mathrm{PS2}}.$$

□

Lemma 2. For a well-locked program $prog$, $[\![prog^?]\!]_{\mathrm{SC}} \subseteq [\![prog]\!]_{\mathrm{SC}}$. Moreover, every machine state reachable by any SC execution of $prog^?$ is reachable by an SC execution of $prog$.

Proof. It is easy to show that every execution of $prog^?$ in SC can be *simulated* by an execution of $prog$ in SC. Whenever `nondetLock` in $prog^?$ fails (*i.e.*, returns 0), `tryLock` in $prog$ also fails. Whenever `nondetLock` in $prog^?$ is trying to get the lock in a loop, $prog$ takes no step. Finally, whenever `nondetLock` in $prog^?$ succeeds to get a lock, `tryLock` in $prog$ can also get the lock since the lock is not acquired by any other thread. □

Lemma 3. For a well-locked program *prog*, we have:

$$[\![prog]\!]_{\mathrm{RA}} \subseteq [\![prog^?]\!]_{\mathrm{RA}}.$$

Proof. For each step *prog* takes, $prog^?$ can simulate the exact step with the following simulation relation. First, the machine state of $prog^?$ and the machine state of *prog* are identical except for the thread views and messages in their memory. Second, the thread views and messages in the memory of $prog^?$ are lower than those of *prog*.

Suppose that the source and the target are related by the above simulation relation. Whenever `tryLock` in *prog* succeeds to acquire a lock, `nondetLock` in $prog^?$ also succeeds. Whenever `tryLock` in *prog* fails, `nondetLock` in $prog^?$ fails. Since `tryLock` in *prog* reads a message and `nondetLock` in $prog^?$ does not, views in the machine state of $prog^?$ remains lower. □

The proof of DRF-Lock-RA (Thm. 7).

Proof. By the definition of RA semantics, the RA-execution of *prog* is *equal* to the RA-execution of $prog[\mathtt{ra}, \mathtt{ra}]$. Thus, if every machine state reachable from a RA-execution of a program *prog* is `rlx`-race-free, then every machine state reachable from a RA-execution of the program $prog[\mathtt{ra}, \mathtt{ra}]$ is `rlx`-race-free as well. Then, we have the following:

$$\begin{aligned}
[\![prog]\!]_{\mathrm{RA}} &= [\![prog[\mathtt{ra}, \mathtt{ra}]]\!]_{\mathrm{RA}} && \text{(by the definition of RA semantics)} \\
&= [\![prog[\mathtt{ra}, \mathtt{ra}]]\!]_{\mathrm{PS2}} && \text{(by Thm. 6)} \\
&= [\![prog]\!]_{\mathrm{PS2}} && \text{(by Lemma 1)}
\end{aligned}$$

□

The proof of DRF-Lock-SC (Thm. 8).

Proof. By Lemma 2, we know that every machine state reachable from $prog^?$ has no race on any non-lock locations. Since $prog^?$ accesses lock locations only using `Lock` and `Unlock`, we can apply DRF-LOCK Theorem in [33].

$$[\![prog]\!]_{\mathrm{SC}} \subseteq [\![prog]\!]_{\mathrm{RA}} \qquad \text{(trivial)}$$

$$\subseteq [\![prog^?]\!]_{\mathrm{RA}} \qquad \text{(by Lemma 3)}$$
$$= [\![prog^?]\!]_{\mathrm{SC}} \qquad \text{(by the DRF-Lock theorem in [33])}$$
$$\subseteq [\![prog]\!]_{\mathrm{SC}} \qquad \text{(by Lemma 2)}$$

Therefore, we have $[\![prog]\!]_{\mathrm{SC}} = [\![prog]\!]_{\mathrm{RA}} = [\![prog^?]\!]_{\mathrm{RA}} = [\![prog^?]\!]_{\mathrm{SC}}$. From Thm. 7 stating $[\![prog]\!]_{\mathrm{RA}} = [\![prog]\!]_{\mathrm{PS2}}$, $[\![prog]\!]_{\mathrm{SC}} = [\![prog]\!]_{\mathrm{RA}} = [\![prog]\!]_{\mathrm{PS2}}$ follows. □

9 Related Work

We have already discussed the challenges in defining a 'sweet-spot' for a programming language concurrency model, which is neither too weak (*i.e.*, it provides programmability guarantees) nor too strong (*i.e.*, it allows efficient compilation). Java was the first language, where considerable effort was put into defining such a formal model [48], but the model was found to be flawed in that it did not permit a number of desired transformations [67]. To remedy this, C/C++ introduced a very different model based on 'per-execution' axioms [8], which was also shown to be inadequate [66, 6, 65, 37]. More recently, PS [33], which has already been discussed at length, addressed this challenge using the idea of locally certifiable promises. PS2 improves PS by supporting inter-thread optimizations and better compilation of RMWs to Armv8. We note that the promise-free fragment of PS2 is identical to the promise-free fragment of PS.

Besides PS, there are three other approaches based on event structures [15, 56, 27]. Pichon-Pharabod and Sewell [56] defined an operational model based on plain event structures. Execution starts with a structure representing all possible program execution paths, and proceeds either by committing a prefix of the structure or by transforming it in a way that imitates a compiler optimization (*e.g.*, by reordering accesses). The model also has a speculation step, whose aim is to capture transformations based on global value range analysis, but has side-condition that is rather difficult to check. The main downside of this model is its complexity, which hinders the formal development of results about it.

Jeffrey and Riely [27] defined a rather different model based on event structures, which constructs an execution via a two player game. The player tries to justify all the read events of an execution, while the opponent tries to prevent him. At each step, the player can extend the justified execution by one read event, provided that for any

continuing execution chosen by the opponent, there is a corresponding write that produced the appropriate value. The basic model does not allow the reordering of independent reads, which means that compilation to Arm and Power are suboptimal. Although the model was later revised to fix the reordering problem [28], optimal compilation to hardware remains unresolved. Moreover, it does not support inter-thread optimizations and/or elimination of overwritten stores, since it forbids the annotated outcome of LB-G (in §11).

Chakraborty and Vafeiadis [15] introduced WEAKESTMO, a model based on *justified* event structures, which are constructed in an operational fashion by adding one event at a time provided it can be justified by already existing events. Justified event structures are then used to extract consistent executions, which in turn determine the possible outcomes of a program. While WEAKESTMO resolve PS's Armv8 compilation problem [52], it does not formally support inter-thread optimizations. Moreover, WEAKESTMO does not support a class of strengthening transformations such as $\mathtt{W}_{\mathtt{rel}} \rightsquigarrow \mathtt{F}_{\mathtt{rel}}; \mathtt{W}_{\mathtt{rlx}}$. Both PS and PS2 support these transformations.

More recently, Java has been extended with different access modes in JDK 9 [38, 39]. Bender and Palsberg [9] formalized this extension with a 'per-execution' axiomatic model similar to RC11 [37]. The model disallows load-store reordering (LB behaviors) for atomic accesses, while allowing out-of-thin-air values for plain accesses. Because of the latter, global value analysis is unsound in this model. It remains unclear, however, whether transformations based on such (unsound) analysis might be sound or not.

10 Discussion

We have presented PS2, the first model that formally enables transformations based on global analysis while supporting programmability (via DRF guarantees and soundness of value-range reasoning) and efficient compilation (including various compiler thread-local optimizations). The inherent tension between these desiderata, together with our goal to have a thread-local small-step operational semantics, naturally leads to a rather intricate model, which is less abstract than alternative declarative models. Nevertheless, we note that PS2, like its predecessor PS, is modeling weak behaviors with just two principles: (*i*) "views" for out-of-order execution of reads; and (*ii*) "promises" for

out-of-order execution of writes. The added complexity of PS2 is twofold: reservations and capped memory. We view reservations as a simple and natural addition to the promises mechanism. Capped memory is less natural and more complex. Fortunately, it is only a part of the certification process and not of normal execution steps. In addition, the DRF-Promise (and the other DRF theorems as well, Theorems 5 to 8) are methods to simplify the semantics. Programmers may safely use the PF or the RA fragment of PS2, which has only views (without any promises, certifications, reservations, or capped memory), when their programs are avoiding data race via release-acquire and lock synchronization.

We also note that PS2 allows some seemingly dubious behaviors, such as *"read from unexecuted branch"* [10]:

$$\begin{array}{l|l} a := X^{\texttt{rlx}} \;\; \textit{// 42} & b := Y^{\texttt{rlx}} \;\; \textit{// 42} \\ Y^{\texttt{rlx}} := a & \texttt{if } b = 42 \texttt{ then} \\ & \quad X^{\texttt{rlx}} := b \\ & \texttt{else} \\ & \quad X^{\texttt{rlx}} := 42 \end{array} \tag{RFUB}$$

The annotated behavior is allowed in PS2 (as in PS and C/C++11). Aiming to support local compiler optimizations, this is actually unavoidable. Practical compilers (including gcc and llvm) may observe that thread 2 writes 42 to X regardless of which branch is taken, and optimize the program of thread 2 to $b := Y^{\texttt{rlx}}; X^{\texttt{rlx}} := 42$ (such optimization is a "trace-preserving transformation" [33]). The resulting program is essentially the LB program (see §11), whose annotated behavior can be obtained by mainstream architectures.

Finally, to the best of our knowledge, PS2 supports all practical compiler optimizations performed by mainstream compilers. As a future goal, we plan to extend it with sequentially consistent accesses (backed up with DRF-SC guarantee) and C11-style consume accesses.

Chapter IV

An In-order Semantics for Relaxed Memory Concurrency

11 Introduction

Recent years have shown multiple proposals of shared-memory concurrency models (see, *e.g.*, [33, 44, 55, 15, 26, 29]). These models typically focus on *performance*, aiming at a semantics that allows various compiler optimizations and efficient mapping to hardware. In particular, to support *load-store reordering* (of accesses to different addresses), either as a part of a compiler optimization or as a possible result of the hardware's pipeline, all these models employ some sort of out-of-order execution that allows reads to read from future writes. For not sacrificing *usability*, which typically means that "out-of-thin-air" values should be forbidden and the model should admit well-accepted data-race-freedom (DRF) guarantees [3, 7, 17], such models have to restrict their speculation mechanisms in a way in which certain program behaviors have to be justified by the existence of other program behaviors. For instance, the promising semantics by Kang et al. [33] requires promises of future writes to be justified by *another* thread-local run of the program, and event-structure models, as the one by Chakraborty and Vafeiadis [15], enforce consistency constraints on a structure that captures *several* runs of the program. This makes these models rather

complex to reason about, and, indeed, besides several notable exceptions for particular models (see, *e.g.*, [64, 2]), existing verification research cannot handle such models.

This chapter is devoted to investigating an alternative approach that puts *amenability to reasoning and verification* in the center. For that, we are after an *in-order semantics*, where each allowed behavior is accounted for by *one* execution of the program in which the actions of the different threads follow the order dictated in their code, and every read reads from a previously executed write. An in-order semantics allows one to incrementally reason about the code line-by-line, considering at each step only the effect of the execution so far and the current instruction. In contrast, reasoning about out-of-order semantics is much harder as it requires considering future instructions (or revisiting previous decisions) based on other possible program executions.

The most intuitive example for an in-order semantics is the well-known model of sequential consistency (SC), where different threads take turns communicating with a single global memory in the form of address-to-value mapping, and every read obtains its value from the last previously executed write to the same address. Nevertheless, various other models, weaker than SC, are still in-order. In particular, RC11 [37], a well-studied declarative model for C/C++ that follows the proposal in [12] to forbid cycles in the union of the program order and the reads-from relation, is an in-order model. Verification for RC11-style models has been extensively studied, and multiple techniques have been developed, including program logics [20, 19, 22], model checkers and fuzzers [34, 35, 47], automatic robustness analyses [49], and library abstraction theorems [60, 63].

Accordingly, our goal is to study: *How far can one go in an in-order semantics?* More concretely, we aim to understand how in-order models can be designed in a way that minimizes the overhead they cause for compiler optimizations and mapping to modern hardware.

We target C/C++ as a source language [11, 8]. Most importantly, this means that programmers distinguish between synchronization accesses (“atomics”) and weak accesses that should not be used for inter-thread synchronization (“non-atomics”), and can cause any behavior when they are misused for this purpose nonetheless. The latter allows us to rely on “undefined behavior” for racy non-atomics, which is a crucial ingredient of our proposed approach. (Thus, we do not provide a solution for

"safe" languages that cannot tolerate undefined behavior.) For atomics, we support the main shared memory constructs of C/C++11, including relaxed and release/acquire accesses, read-modify-writes, and release/acquire and sequentially consistent fences.

Non-atomic accesses account for the vast majority of memory accesses in concurrent programs, while atomics, which are used for inter-thread communication and synchronization, are relatively rare. In particular, among atomics, the only ones that are intended to allow the problematic load-store reordering are relaxed accesses, which are meant to be used by "very careful" programmers [11] and are often confined to libraries that are manually optimized by experts. Thus, we believe that the trade-off between performance and amenability to reasoning should be investigated differently for atomics and non-atomics. Next, we separately discuss the performance overhead that is imposed by an in-order semantics for supporting non-atomic accesses and atomic accesses.

Overhead in Non-atomic Accesses Our first question is whether it is possible to have an in-order semantics without imposing any performance overhead for non-atomic accesses. This stems from a principled approach: being non-racy, non-atomics should allow all compiler optimizations that are performed in single-threaded code.[1] We observe that a significant challenge exists for validating this guiding principle in an in-order model, and we are not aware of any existing model that solves this challenge (even for a simple fragment with only non-atomics and strong synchronization accesses with, say, release/acquire semantics). In particular, RC11 invalidates (irrelevant) *load introduction*, a transformation widely used in sequential code with significant possible performance gains. In fact, the LLVM manual requires that non-atomics should validate all optimizations allowed on sequential accesses (the only exception is store introduction, which compilers avoid also in sequential code), and explicitly mentions that load introduction may be performed by the compiler, and the LLVM compiler indeed introduces non-atomic loads as a part in several of its optimization

[1] Compiler optimizations on single-threaded code effectively cover modern hardware's behaviors of plain loads and stores, so it is sufficient to focus this discussion on validating compiler optimizations. Still, in our results, we prove the correctness of mapping non-atomics to plain machine loads and stores.

```c
extern void foo(unsigned int* x);

unsigned int test(unsigned int n) {
  unsigned int x[1], sum = 0;
  foo(x);
  for (unsigned int i = 0; i < n; i++)
    sum += x[0];
  return sum;
}
```

(a) Before optimization

```c
extern void foo(unsigned int* x);

unsigned int test(unsigned int n) {
  unsigned int x[1], sum = 0;
  foo(x);
  sum = x[0] * n;

  return sum;
}
```

(b) After optimization

Figure IV.1: An example of load introduction. The program on the left adds the value in `x[0]` `n` times. GCC 12.2.0 with `-Os` flag and Clang 15.0.0 with `-O2` flag compile this program into the one on the right (written in C instead of assembly for readability) by turning the loop into a multiplication. This optimization effectively introduces a load from `x[0]` when `n` = 0.

passes.[2] The assumptions of the GCC compiler are less clear, but some examples show that it introduces loads as well. A concrete example is given in Fig. IV.1.

We provide a full solution to this challenge, and design an in-order semantics that does not sacrifice any optimization on non-atomics. Inspired by LLVM, the key to doing so is to utilize the distinction between a *source* semantics and an *intermediate representation* (IR) semantics. This allows the separation of concerns: compiler optimizations may be unsound in the source semantics, whereas the IR semantics does

[2]See https://llvm.org/docs/Atomics.html#optimization-outside-atomic [Accessed November 2022].

not have to be in-order. Indeed, the IR is not meant to be amenable to conventional verification and reasoning, and programmers in the source language only need to know the source semantics. This strategy, however, is not a magic potion: to have a sound compilation, these models have to be designed in a way that the IR semantics is stronger than the source semantics (*i.e.*, all behaviors allowed by the IR should be allowed by the source).[3]

Our main contribution is to show that this approach works with the right choice of source and IR. Concretely, we develop an in-order source model, based on the *promise-free* fragment of the promising semantics, and an IR model based on a recent version of the promising semantics in [18], and prove the required relation between them. Our proposed source model is (slightly) stronger than RC11, which allows the application of previous work on verification under RC11. (In particular, we observe that certain races on non-atomics can be safely ignored in RC11's catch-fire mechanism.) For the IR, we have ported the result of [18], which establishes the correctness of all optimizations on non-atomics that are allowed in sequential code. This means that most compiler optimizations can be formally validated based on sequential reasoning, so even most compiler developers need not understand the out-of-order IR model.

We note that while we mostly employ existing models (with some modifications and simplifications), to the best of our knowledge, this work is the first to formally relate an in-order source model and an out-of-order IR model with the goal of having an in-order source semantics without any performance overhead for non-atomics.

Overhead in Atomic Accesses Naturally, the next question is about the performance overhead for atomic accesses. Here, the challenge concerns *relaxed* accesses, which are meant to allow load-store reordering that is in sharp contrast with in-order semantics. Unfortunately, we show that any in-order model that supports all optimizations on non-atomics has to forbid the reordering of a non-atomic/relaxed read followed by a relaxed write. (In particular, this reordering is forbidden in both the

[3]It is sufficient to have a correct efficient mapping from the source to the IR, where correctness is in the standard sense: every behavior that is allowed by the IR semantics (of the mapped program) is also allowed by the source semantics (of the source program). Since we do not want to sacrifice any performance in this mapping, we actually consider this mapping being the *identity mapping*, and, thus, we simply require that the IR semantics is stronger than the source.

source and the IR models we propose.)

What is the practical impact of forbidding this reordering? First, we note that although compiler optimizations that reorder and eliminate atomics were extensively studied before (see, *e.g.*, [65, 21]), to the best of our knowledge, existing compilers do not perform any of these optimizations. Then, it remains to understand the implications on the mapping to hardware. Indeed, load-to-store ordering between plain accesses is not guaranteed to be preserved by existing models of modern architectures, like those of Arm [58, 4] and Power [61, 5],[4] and so, forbidding this reordering seems to require a stronger mapping of relaxed accesses for these architectures.

Interestingly, we observe a significant gap between CPU *models* and observable behaviors *in practice* regarding the preservation of load-store ordering. While the abstract models of Arm and Power allow the reorder of loads followed by stores, such behaviors were observed only in very few CPU implementations.[5] In our discussion with CPU architects from Arm, we confirmed that the load-store reordering is explicitly prohibited in Cortex processors, starting from Cortex-A76. From this discussion, we further understood the technical trade-offs involved in their design, and learned that, compared to other possible reorderings that the hardware performs, load-store reordering is hard to apply and has rather limited performance benefits.

Accordingly, we propose a practical approach to this challenge. In the long term, we believe the right way to go is for vendors to introduce new kinds of store instructions, which we call "strong stores", and officially preserve the order from loads to strong stores.[6] We expect a minimal (to no) overhead for these instructions compared to plain stores. In particular, strong stores still admit store-store reorderings, which are commonly observed in practice, and are thus weaker than release stores that are more expensive to implement. Meanwhile, in the absence of such instructions, we

[4]Intel's architecture (assuming x86-TSO by Owens et al. [54]), has rather strong semantics for plain loads and stores, which never reorders loads with later stores.

[5]Load-store reordering (concretely, the weak behavior of the LB litmus test) was never observed on Power as well as on various implementations of Armv8 that were tested in [5, 4]. An anonymous review of this paper provided information showing that this reordering is observed on Cortex A73, and mentioned that even on Cortex post A76 load-store reordering can be observed when memory locations are mapped to device, or when vector instructions are used.

[6]Alternatively, we may introduce a load-store fence that orders all preceding loads with succeeding stores. We discuss the benefits of using load-store fences instead of strong store instructions in §16.4.

propose to compile relaxed writes differently depending on the target hardware: (*i*) for a target that preserves load-store order, the compilation can use plain accesses; and (*ii*) otherwise, relaxed writes have to be compiled as release writes.

Outline The rest of this chapter is structured as follows. In §12, we present the challenges, key ideas, and observations of this chapter in more detail. In §13, we present (a simplified fragment of) the proposed source model and discuss its relation to RC11. In §14, we present (a simplified fragment of) the IR model and establish the soundness of mapping the source model to the IR model. In §15, we present the full source and IR models. In §16, we discuss the mapping to modern hardware, its soundness, and the proposed additions to hardware models. In §17, we provide the pen-and-paper proofs of our results on relating the source model to declarative models. Finally, in §18, we discuss related work.

Supplementary Material Our main results (1. soundness of mapping from source to IR, 2. soundness of mapping from IR to Armv8, 3. DRF guarantees for the source, and 4. adequacy of sequential reasoning for validating optimizations in the IR) are **mechanized in Coq**. The supplementary material available online [41] includes the Coq development and the results of our experiments.

12 Challenges and Key Ideas

In this section, we present more details on the main observations and contributions of this chapter. To a significant extent, our central contributions are not in developing new concurrency models and proving their meta-theoretic properties but rather in providing a holistic analysis and approach to the problem of a shared-memory concurrency semantics in a high-level language like C, C++, or Rust. Like in §11, we separately discuss non-atomics (§12.1) and atomics (§12.2) while focusing on compiler optimizations for non-atomics and mapping to hardware for atomics. (Our results include the mapping of non-atomics to plain accesses on hardware, as well as compiler optimizations involving atomics, but these are not discussed in this section.)

12.1 Optimizing Non-Atomics in an In-Order Semantics

Supporting sequential optimizations for non-atomics in an in-order semantics is highly challenging, and, to the best of our knowledge, it was not addressed by previous work. Next, we demonstrate the challenge using the well-known load buffering example and how we address this challenge.

Read-Write Reordering vs. In-Order Semantics

To understand the crux of the challenge, consider the classical example below, known as the load buffering litmus test with non-atomic accesses (LB-NA, for short), where all accesses are marked as non-atomics (`na`).

$$\begin{array}{l|l} a := X^{\mathtt{na}} & b := Y^{\mathtt{na}} \\ Y^{\mathtt{na}} := 1 & X^{\mathtt{na}} := 1 \\ \mathtt{print}\ a & \mathtt{print}\ b \end{array} \qquad \text{(LB-NA)}$$

Here and henceforth, we assume that all variables are implicitly initialized to 0. Our requirement on compiler optimizations implies that the behavior in which both threads printing 1 must be allowed. Indeed, the compiler may reorder the read from X and the write to Y in the first thread (this is certainly possible in sequential code, thus non-atomics should allow the reordering as well), and then $a = b = 1$ is possible even under SC. This behavior is in tension with the requirement to have an in-order semantics for the source language, which will have to execute one of the reads first, and at that point the only available write to read from is the implicit initialization write of the value 0.

Catch-Fire as a Solution?

A well-known approach to address the above example is to exploit the fact that non-atomics are not supposed to be used for inter-thread synchronization and avoid providing any guarantees on the program behaviors when non-atomics participate in data races. This idea, which we refer to as "catch-fire" semantics, is the cornerstone of the C/C++11 [8], and its repaired version RC11 [37], which explicitly states that a

data race on non-atomic accesses implies *undefined behavior* (UB, for short) for the given program.

Accordingly, RC11 allows the annotated behavior of the LB example above, while still being an in-order semantics. A particular run, for instance, could perform both memory accesses of the first thread (read 0 and write 1), observe a forbidden data-race when executing the first (or second) access of the second thread, and then invoke UB. In turn, UB allows any possible continuation of the execution, which in particular includes the ability to print 1 by both threads. This is still an in-order semantics: threads execute their actions in the order specified by the program, a data-race is detected according to *previously* executed accesses, and UB only affects future decisions.

Remark 6. The original presentation of RC11 in [37] identifies program behaviors with "final outcomes" (mapping each variable to the modification-order-maximal value written to it). The current discussion assumes that behaviors are captured by sequences of system calls (*e.g.*, results of print statements) generated by a given program. RC11 can be easily adapted to this notion by assuming that consistent execution graphs are incrementally constructed during the program run, system calls are observed in the order they were executed along the run, and any suffix of system calls is allowed once a racy execution graph is reached.

Load Introduction

A catch-fire semantics validates various compiler optimizations on non-atomics, including access reordering and redundant access elimination. Indeed, whenever such transformations enable additional behaviors, it can be shown that the source program was already racy, and justify the target behaviors by UB invoked by the source. Catch-fire, however, falls short to *fully admit* our guiding principle: some transformations allowed on sequential code are still disallowed on non-atomics.

Concretely, the problem is with (irrelevant) *load introduction*. If the effect of the compiler's optimization introduces a non-atomic load (which may happen, *e.g.*, when transforming

$$\texttt{while}\ B\ \texttt{do}\ \{a := X^{\texttt{na}};\ \ldots\} \qquad \text{to} \qquad a := X^{\texttt{na}};\ \texttt{while}\ B\ \texttt{do}\ \{\ldots\}$$

in traces where B evaluates to $false$), then the target program may be racy (and invokes UB), while the source is not. Thus, any model based on catch-fire mechanism cannot validate load introduction.

Load introduction is necessary for multiple optimizations based on speculation, which are commonly performed by compilers (Clang, in particular) when hoisting loads, *e.g.*, as a part of loop invariant code motion, loop unswitching, load-widening or when loading a vector while only a subset of elements is needed.[7]

In addition to Fig. IV.1 from §11, Fig. IV.2 demonstrates another case where load introduction has the potential to significantly improve performance. In Fig. IV.2, the program on the left stores `x[j]` into `y[j]` for each odd `j` (and `0` otherwise). The program on the right is a hand-optimized version: it introduces loads from `x[0]`, `x[2]`, `x[4]`, and `x[6]`; merges all loads into a single 8 bytes load; and stores the result with an appropriate mask into `y` by a single 8 bytes store. Here, external functions (`foo` and `bar`) are used only to prevent the compiler from eliminating the loads and stores. By compiling both programs with Clang 15.0.1 and running them on ThunderX2 Armv8 server, we observed more than x2 performance gain (average execution time of 0.069s vs. 0.033s).

Undefined Value as a Solution?

A natural idea for supporting load introduction is to limit the "undefinedness" to the value being read in racy reads: instead of invoking UB, just leave unspecified the value loaded by a non-atomic racy read, so if this value is never used (and the load is indeed irrelevant), we will not introduce additional behaviors. The LLVM semantics follows this idea: it keeps read-write races to be always well-defined and declares that non-atomic racy reads may return "undef" value. In turn, "undef" can be refined to *any* value.[8]

While being tempting at first sight, undefined value for racy reads will not solve our problem. Referring back to the LB example above, it is easy to see that any execution of an in-order semantics can observe a race only in one of the reads, so only one of

[7] See `https://llvm.org/docs/Passes.html` [Accessed November 2022].

[8] Branching on "undef" is still considered UB. The "freeze" instruction recently introduced in LLVM is a tool to support branching on a possibly undefined value, which is often a result of load introduction [40].

```c
extern void foo(char* x), bar(char* x);

int main() {
  char x[8], y[8];

  for (int i = 0; i < 10000000; i++) {
    foo(x);
    for (int j = 0; j < 8; j++)
      y[j] = j%2 ? x[j] : 0;
    bar(y);
  }

  return 0;
}
```

(a) Before optimization

```c
extern void foo(char* x), bar(char* x);

int main() {
  char x[8], y[8];

  for (int i = 0; i < 10000000; i++) {
    foo(x);
    uint64_t r = *(uint64_t*)x;
    *(uint64_t*)y = r & 0xFF00FF00FF00FF00ul;
    bar(y);
  }

  return 0;
}
```

(b) After optimization

Figure IV.2: A hand-optimized example of load introduction.

them can return "undef", which will not allow *both* threads to print 1. To fix this, one has to either speculate a data race when performing the first read, or revisit its previous decisions on the read value when performing the second write. Both options lead us to models that are much more complicated than in-order models.

Our Proposal: An Intermediate Representation

The key idea in our approach is to split the semantics into two models: a source model that accounts for the programmers' needs, and an intermediate representation (IR) model that accounts for the compilers' (and hardware's) needs. Then, the compiler first maps the source program to the IR, and only then applies its optimizations. Programmers should be only aware of the source model, which can be in-order (*e.g.*, with catch-fire) since it does not have to support compiler optimizations; and the IR semantics can support compiler optimizations (*e.g.*, with undefined value for racy reads and out-of-order race detection) since it does not have to be in-order. Our manifestation of this approach consists of the following contributions:

1. We propose a source model, which we denote by vRC11, obtained by adding non-atomic accesses to the *promise-free* fragment of the promising semantics [33, 44]. This model, which is stronger than RC11, is formulated as an operational model using timestamps and thread-views to justify weak behaviors, and a simple race-detection mechanism that invokes UB for races on non-atomics. Interestingly, we observe that not all such races should invoke UB, and it is sufficient to consider races with previously executed writes (and ignore races with previously executed reads). Thus, we are able to restrict the catch-fire mechanism in a way that deems fewer programs racy but still achieves what catch-fire is needed for.

2. For the IR model, we develop a simplification of the promising model by Cho et al. [18], which we denote by PS^{IR}, where (simplified) promises are only needed for race detection. Thus, PS^{IR} justifies an out-of-order behavior by detecting a race with "promises" made by other threads. More concretely, a thread in PS^{IR} can *promise* to execute a non-atomic write to a location X in the future, whenever the thread can *certify* the promise by checking that it can perform a non-atomic write to X by executing alone. Once a promise is made, another thread reading from X races with the promise and reads "undef" value. We have ported the result of [18] to PS^{IR}, which establishes the correctness of all optimizations on non-atomics that are allowed in sequential code.

3. We prove that PS^{IR} is stronger than vRC11. Roughly speaking, this is possible

because catch-fire is sufficiently weak to account for the IR's out-of-order behaviors. In other words, once a program exhibits any behavior that stems from an out-of-order execution under PS^{IR}, the same program has a race in a (possibly different) execution under vRC11, where a race leads to UB. To establish the proof, it is enough to show that the source can invoke UB for such an out-of-order execution in PS^{IR}. Here, the key idea is that the thread of vRC11 can follow the certification run (which is required to justify a promise under PS^{IR}) and perform a non-atomic write to X instead of making a promise to X. Then, the other thread reading from X races with that non-atomic write, and the program invokes UB under vRC11.

Example 1. In the LB-NA example, PS^{IR} allows $a = b = \texttt{undef}$ through an out-of-order execution where the first thread promises to write to Y, and the second thread reads "undef" from Y since the read races with the promise. The first thread could certify its promise before making it by reading 0 from X and executing $Y^{\texttt{na}} := 1$. In vRC11, instead of promising the write, the first thread can execute and write to Y following the certification execution of PS^{IR}. Then, the second thread's read from Y becomes racy, and the program invokes UB, which accounts for all possible behaviors.

Remark 7. In fact, since UB by the source accounts for any behavior of PS^{IR}, the proof of mapping the source to the IR can essentially assume that there is no race in the promise-free fragment of PS^{IR}, which makes the mapping proof similar to the proof of the DRF-PF theorem (a data-race-freedom guarantee w.r.t. the promise-free semantics) in [17].

We note that the fact that PS^{IR} is stronger than vRC11 allows one to soundly reason about programs under PS^{IR} semantics while assuming vRC11 (which may be needed when the intermediate language itself acts as a source language for another step of compilation and thus is not completely compiler-internal). Such reasoning would be incomplete, but we expect that only a small fraction of programs will need a precise analysis using the exact IR model.

12.2 Mapping Relaxed Accesses to Modern Hardware

In this section, we turn to the question of supporting atomic accesses focusing specifically on relaxed accesses. We first demonstrate the challenge and propose two practical

solutions, a long-term solution that depends on hardware vendors implementing our feature request and a short-term solution that requires strengthening the existing compiler mapping of relaxed accesses for certain hardware implementations.

Reordering of Relaxed Accesses in an In-Order Semantics

Like non-atomic accesses, relaxed accesses in C/C++11 were intended to be mapped to plain loads and stores in the hardware even when the hardware model allows load-store reordering. Clearly, this is in contrast with an in-order semantics (indeed, consider the LB example above with relaxed accesses). Moreover, since relaxed accesses are meant to be used in races (for improving the performance of certain concurrency idioms; see, *e.g.*, [62]), catch-fire is not a possible solution here. In fact, as the next example shows, even the reordering of a non-atomic load followed by a relaxed atomic store cannot be allowed in an in-order semantics:

$$\begin{array}{l} a := Y^{\mathtt{rlx}} \\ \mathtt{if}\ a = 1\ \mathtt{then} \\ \quad X^{\mathtt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} \mathtt{if}\ *\ \mathtt{then} \\ \quad b := X^{\mathtt{na}} \\ \quad \mathtt{if}\ b = 1\ \mathtt{then}\ Y^{\mathtt{rlx}} := 1 \\ \quad \mathtt{print}\ b\ \ \textit{// prints 1} \\ \mathtt{else} \\ \quad Y^{\mathtt{rlx}} := 1 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := Y^{\mathtt{rlx}} \\ \mathtt{if}\ a = 1\ \mathtt{then} \\ \quad X^{\mathtt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} Y^{\mathtt{rlx}} := 1 \\ c := X^{\mathtt{na}} \\ \mathtt{if}\ c = 1\ \mathtt{then} \\ \quad \mathtt{print}\ 1 \end{array}$$

(LB-CHOICE)

Here, "$*$" means a *non-deterministic choice* that non-deterministically returns arbitrary value.[9] Assuming an in-order semantics, the source program on the left cannot print 1 since either one of $a := Y^{\mathtt{rlx}}$ or $b := X^{\mathtt{na}}$ executes first and can only read 0 (from the initial memory). However, as shown in Fig. IV.3, by applying a sequence of compiler transformations on non-atomics in the second thread, the program on the left can be transformed to the program on the right. Then, if the reordering of $c := X^{\mathtt{na}}$ and $Y^{\mathtt{rlx}} := 1$ is allowed (by the compiler or the target hardware), the second thread printing 1 is easily observable (even under SC). Therefore, the reordering of a non-atomic load followed by a relaxed store must be *forbidden* in

[9] A non-deterministic choice corresponds to "freezing" an undefined value in LLVM. See https://llvm.org/docs/LangRef.html#undefined-values and https://llvm.org/docs/LangRef.html#freeze-instruction [Accessed November 2022].

(1)	(2)	(3)	(4)	(5)
$c := X^{\mathtt{na}}$	$c := X^{\mathtt{na}}$	$c := X^{\mathtt{na}}$	$c := X^{\mathtt{na}}$	$c := X^{\mathtt{na}}$
				$Y^{\mathtt{rlx}} := 1$
`if *` `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`
$b := X^{\mathtt{na}}$	$b := X^{\mathtt{na}}$	$b := 1$	$b := 1$	
`if` $b = 1$ `then`	`if` $b = 1$ `then`	`if` $b = 1$ `then`		
$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	
`print` b	`print` b	`print` b	`print` 1	`print` 1
`else`	`else`	`else`	`else`	
$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	$Y^{\mathtt{rlx}} := 1$	

(1) introduce a non-atomic read $c := X^{\mathtt{na}}$;

(2) replace the non-deterministic choice with an expression;

(3) forward the read $c := X^{\mathtt{na}}$ to the read $b := X^{\mathtt{na}}$, turning it into $b := 1$;

(4) forward $b := 1$ to the expression $b = 1$ and the print statement; and

(5) hoist the common write $Y^{\mathtt{rlx}} := 1$ out of the branch.

Figure IV.3: A sequence of compiler transformations on non-atomics applied to the second thread of LB-CHOICE.

any in-order source semantics that aims to allow common compiler transformations on non-atomics.

Revisiting the Assumptions on Hardware

We observe that there is a significant gap between CPU *models* and observable behaviors *in practice* regarding the preservation of load-store ordering. While the abstract models effectively allow the reordering of loads with subsequent stores, such behaviors are rarely observed in practice. Indeed, previous experiments performed to validate the hardware models rarely observed weak behaviors of the LB litmus test. First, such behaviors were never observed on any Power hardware [61, 5]. Second, while they were observed on several Armv7 implementations, to the best of our knowledge, for Armv8, LB was only observed on Qualcomm's Snapdragon 820 mobile processors [4]

and on Cortex A73.[10] To gain more confidence, we experimentally tested a newer version, Qualcomm's Snapdragon 888 processor, and the weak behaviors of LB were not observed there.

After discussing with Arm engineers, we gained a better understanding of the architectural reasons why the potential performance improvement by allowing load-store reordering is relatively small. Essentially, this stems from the fact that a store can be treated as completed in its own core when it is added to the core-local store buffer before being made visible to other cores. Thus, no intra-core optimization is prevented by preserving load-store ordering. The only exception is that such reordering may reduce the pressure on the store buffer (so that fewer stores stall due to the buffer being full), as it allows to commit a store from the store buffer to the shared storage possibly before previous loads were completed. However, committing stores early complicates the cache implementation regarding the ECC (error correction code) logic, and before committing a store, the core must check that all incomplete preceding loads will never raise exceptions and are not aliased with the store to be committed.

Remark 8. Unlike load-store ordering, preserving store-store ordering is rather expensive. For instance, in Cortex A76 and later versions, a store from the store buffer is committed to a *merge buffer* when it is the oldest store (*i.e.*, all preceding loads are completed, and all preceding stores are already committed to the merge buffer). Then, stores in the merge buffer may be reordered to group together those writes that fit in the same cache line, which are merged and committed at once. Such reordering between stores greatly reduces cache accesses and is thus considered performance-critical, which is why store-store reordering visible to other cores is needed.

A Long Term Practical Solution

Based on the above discussion, we raise a clear "feature request" from hardware vendors. Concretely, we propose hardware vendors to introduce a new kind of store instructions, which we call "strong stores", that will preserve load-store ordering. Then, the IR's relaxed stores will be mapped to strong hardware stores. For most

[10] Snapdragon 820 exhibits various other weak behaviors that are forbidden by the official model (950 such tests reported in [4]!). The information about Cortex A73 was obtained from the anonymous PLDI reviewer.

hardware architectures, where architects agree that load-store ordering is preserved, strong stores could be implemented as plain stores. Otherwise, the overhead is not expected to be significant, and, in any case, strong stores should be cheaper than release stores (since they do not need to preserve store-store order).

We believe that this is a case where the input from multiple years of research in concurrent programming language semantics may guide hardware developers. In fact, other features of Arm, such as sequentially consistent accesses and release sequences, were developed hand in hand with C/C++11 constructs. Our proposal is of a similar nature, identifying an opportunity for hardware vendors to significantly assist programming language design with a rather minimal cost.

In §16, we provide the proposed formal additions to the declarative models of Armv8 and Power for supporting strong stores. We have performed extensive validation of these revised models using the Herd model checker [4, 5], to see that, indeed, when strengthening all stores to be strong, the behaviors that become disallowed are, like LB, behaviors that were not observed on hardware (except for Snapdragon 820 and Cortex A73 as discussed above).

A Short Term Practical Solution

Without the availability of "strong stores" in hardware, we propose to change the compiler mappings to take into account the target CPU. For CPUs that preserve load-store ordering, it is still safe to map relaxed accesses to plain accesses. Otherwise, the compiler should map relaxed store as it maps *release* stores (*e.g.*, to an `stlr` instruction on Armv8).

Following Ou and Demsky [53], mapping relaxed stores as release entails a performance overhead of 3.6% on Arm (although it is rather hard to estimate performance for real-world programs). We note that the mapping scheme that enforces the preservation of load-store ordering by inserting a (fake) branch from every relaxed read, which is more efficient according to Ou and Demsky [53] (with -0.3% overhead), is *unsound* for our needs. Indeed, as the LB-CHOICE example shows, we also need to forbid reordering of *non-atomic* reads followed by relaxed writes. This would require adding a branch from every non-atomic read, which, given the prevalence of non-atomic reads in concurrent programs, is expected to significantly harm performance.

13 The Source Model

In this section, we present the in-order source semantics vRC11 (standing for "view-based RC11"), which we obtain by adding transitions for non-atomic accesses to the promise-free fragment of the promising semantics (PS, for short). In §13.1, we discuss the relation between vRC11 and RC11 and show that vRC11 is stronger than RC11. Therefore, verification theory and tools developed for RC11 (or any weaker model), such as model checkers [35, 34, 47], program logics [19, 23], and robustness analysis [36], all apply to vRC11. In §13.2, we provide a declarative presentation of the model.

vRC11 is obtained from PS by (i) removing the notion of *promises* that models early execution of writes and all transitions and components of states related to promises; and (ii) adding transitions for non-atomic and racy accesses. Next, we introduce the fragment of vRC11 consisting of non-atomic, relaxed and release/acquire writes and reads. In turn, read-modify-writes (RMWs), fences, and release sequences are omitted by brevity. They are included in the full model in Coq and presented in §15.1. Figure IV.4 summarizes the domains and the transitions of vRC11, highlighting the differences w.r.t. the promise-free fragment of the model in [33].

Program Semantics We assume that the program of each thread is represented as a labeled transition system, whose states, denoted by σ, record the local register file and the continuation code, and transitions $\sigma \xrightarrow{l} \sigma'$ are labeled with the action l that is performed. For silent transitions that do not communicate with the memory (*e.g.*, conditionals and local assignments), we write $\sigma \to \sigma'$. Read and write transitions have labels $l = \mathsf{R}(X, o_{\mathsf{R}}, v)$ and $l = \mathtt{W}(X, o_{\mathtt{W}}, v)$, respectively. We also assume transitions executing *system calls*, which are externally observable (*e.g.*, resulting from print statements), with a label $l = \mathtt{Sys}(e)$ where e is the output of the call.

Memory A *memory* M is a finite set of *messages* of the form $m = \langle X@t, v, o_{\mathtt{W}}, V_{\mathtt{m}} \rangle$ representing a previously executed write of a value $v \in \mathsf{Val}$ to a location $X \in \mathsf{Loc}$. Each message has a *timestamp* $t \in \mathsf{Time}$, where Time is the set of non-negative rational num-

$v \in \mathsf{Val}$	value	$m = \langle X@t, v, o_{\mathtt{W}}, V\rangle \in \mathsf{Msg}$	message
$X, Y, Z \in \mathsf{Loc}$	location	$M \subseteq \mathsf{Msg}$	memory
$o_{\mathtt{R}} \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}\}$	read access mode	σ	program state
$o_{\mathtt{W}} \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}\}$	write access mode	$T = \langle \sigma, V\rangle \in \mathsf{Lts}$	thread state
$\tau \in \mathsf{Tid} \triangleq \{\tau_1, \tau_2, \ldots\}$	thread identifier	$\langle T, M\rangle$	thread configuration
$t \in \mathsf{Time} \triangleq \{0\} \cup \mathbb{Q}^+$	timestamp	$\mathcal{T} \in \mathsf{Tid} \to \mathsf{Lts}$	thread state mapping
$V \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$	view	$\mathcal{M} = \langle \mathcal{T}, M\rangle$	machine state

(SILENT)
$$\frac{\sigma \to \sigma'}{\langle\langle \sigma, V\rangle, M\rangle \to \langle\langle \sigma', V\rangle, M\rangle}$$

(SYSTEM CALL)
$$\frac{\sigma \xrightarrow{\mathsf{Sys}(e)} \sigma'}{\langle\langle \sigma, V\rangle, M\rangle \xrightarrow{\mathsf{Sys}(e)} \langle\langle \sigma', V\rangle, M\rangle}$$

(RACE)
$$\frac{\langle X@t, \_, o_{\mathtt{W}}, \_\rangle \in M \qquad V(X) < t \qquad o_{\mathtt{W}} = \mathtt{na} \vee o = \mathtt{na}}{\mathsf{race}(V, M, X, o)}$$

(READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_{\mathtt{R}}, v)} \sigma' \qquad \langle X@t, v, \_, V_{\mathtt{m}}\rangle \in M \qquad V(X) \le t \qquad V' = V[X \mapsto t] \sqcup \begin{cases} 0 & o_{\mathtt{R}} \neq \mathtt{acq} \\ V_{\mathtt{m}} & o_{\mathtt{R}} = \mathtt{acq} \end{cases}}{\langle\langle \sigma, V\rangle, M\rangle \to \langle\langle \sigma', V'\rangle, M\rangle}$$

(WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_{\mathtt{W}}, v)} \sigma' \qquad m = \langle X@t, v, o_{\mathtt{W}}, V_{\mathtt{m}}\rangle \qquad V(X) < t \qquad M \# m \qquad V' = V[X \mapsto t] \qquad V_{\mathtt{m}} = \begin{cases} \lambda X.\, 0 & o_{\mathtt{W}} \neq \mathtt{rel} \\ V' & o_{\mathtt{W}} = \mathtt{rel} \end{cases}}{\langle\langle \sigma, V\rangle, M\rangle \to \langle\langle \sigma', V'\rangle, M \cup \{m\}\rangle}$$

(RACY-READ/WRITE)
$$\frac{l \in \{\mathsf{W}(X, o, \_), \mathsf{R}(X, o, \_)\} \qquad \sigma \xrightarrow{l} \_ \qquad \mathsf{race}(V, M, X, o)}{\langle\langle \sigma, V\rangle, M\rangle \to \langle\langle \bot, V\rangle, M\rangle}$$

(MACHINE: NORMAL)
$$\frac{\langle \mathcal{T}(\tau), M\rangle \xrightarrow{l} \langle T', M'\rangle}{\langle \mathcal{T}, M\rangle \xrightarrow{l} \langle \mathcal{T}[\tau \mapsto T'], M'\rangle}$$

(MACHINE: UB)
$$\frac{\langle \mathcal{T}(\tau), M\rangle \to \langle\langle \bot, \_\rangle, M'\rangle}{\langle \mathcal{T}, M\rangle \to \langle \bot, M'\rangle}$$

Figure IV.4: Domains and transitions of vRC11 (RMWs, fences, and release sequences are omitted). Differences w.r.t. the promise-free fragment of PS are highlighted.

bers,[11] a *write access mode* $o_{\mathtt{W}}$ of the operation by which the message was added, and a *view* $V_{\mathtt{m}} \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$ for enabling release/acquire synchronization, which we explain below. The initial memory consists of an initial message $\langle X@0, 0, \mathtt{na}, \lambda X.\, 0\rangle$ for every location X.

[11] As in previous work [33, 44], timestamps are densely ordered, so one can always add a message between existing messages. This property is particularly useful when proving the soundness of compiler transformations such as "store merge" that merges two successive stores $X := 1 ; X := 2$ into a single store $X := 2$. In the proof, the source has to mimic the target program by finding a free timestamp for $X := 1$ before $X := 2$.

States A *machine state* $\mathcal{M} = \langle \mathcal{T}, M \rangle$ consists of a function $\mathcal{T}$ assigning a thread state to each thread identifier, and a memory M shared among the threads. A *thread state* is a pair $T = \langle \sigma, V \rangle$ where σ is a local program state and $V \in \mathsf{View}$ is a *thread view*, recording the latest timestamp that has been observed the thread for each location. The initial thread state consists of the initial program state and the 0-view assigning 0 to each location.

Read Step A thread can read a message $\langle X@t, v, o_{\mathtt{W}}, V_{\mathtt{m}} \rangle \in M$ with a timestamp greater than or equal to the thread's view of X (*i.e.*, $V(X) \leq t$), updating its view of X to include the timestamp t of the message. If the read is an *acquire* (`acq`) read, the thread also *acquires* the message view $V_{\mathtt{m}}$ and joins it to its own view by taking pointwise maximum (denoted by $\sqcup$).

Write Step A thread writes by adding a message $m = \langle X@t, v, o_{\mathtt{W}}, V_{\mathtt{m}} \rangle$ to the memory M provided that t is greater than the thread's view ($V(X) < t$) and that there is no existing message in M with location X and timestamp t (denoted by $M\#m$). The access mode $o_{\mathtt{W}}$ of the write operation is recorded in m. The thread updates its view to $V' = V[X \mapsto t]$. A release write records the thread's view ($V_{\mathtt{m}} = V'$) in the message, while non-release writes have the 0-view in $V_{\mathtt{m}}$.

Racy Access A memory access to location X by a thread with view V is *racy* if there is some message $\langle X@t, v, o_{\mathtt{W}}, V_{\mathtt{m}} \rangle \in M$ with $V(X) < t$ and either the message is written by a non-atomic write ($o_{\mathtt{W}} = \mathtt{na}$) or the access itself is non-atomic (as defined in (RACE) in Fig. IV.4). Executing a racy read or a racy write leads the thread to the $\bot$ program state.

Machine Step Machine steps are obtained as standard interleaving of thread steps $\langle \mathcal{T}(\tau), M \rangle \to \langle T', M' \rangle$. If the thread detects a race and steps to $\bot$, the machine may take a (PF-MACHINE: UB) step that leads to the $\bot$ machine state, that is later interpreted as UB.

Behavior An (observable) *behavior* is a sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$ of system calls. A machine state $\mathcal{M}$ *generates* a behavior s, denoted by $\mathcal{M} \Downarrow s$, if s is obtained by

restricting a trace of vRC11 starting from $\mathcal{M}$ to system call labels and replacing UB by an arbitrary suffix of system calls. With standard notations for sequences, $\mathcal{M} \Downarrow s$ is defined by:

$$\frac{\mathsf{terminal}(\mathcal{M})}{\mathcal{M} \Downarrow \epsilon} \qquad \frac{\mathcal{M}_1 \to \mathcal{M}_2 \qquad \mathcal{M}_2 \Downarrow s}{\mathcal{M}_1 \Downarrow s} \qquad \frac{\mathcal{M}_1 \xrightarrow{\mathsf{Sys}(e)} \mathcal{M}_2 \qquad \mathcal{M}_2 \Downarrow s}{\mathcal{M}_1 \Downarrow e \cdot s} \qquad \frac{\mathcal{M} \to \langle \bot, \_ \rangle}{\mathcal{M} \Downarrow s}$$

Here, $\mathsf{terminal}(\mathcal{M})$ means that the machine state $\mathcal{M}$ is terminal (*i.e.*, every thread has empty continuation code). As captured by the last rule, once a UB is invoked during the execution, the machine exhibits any behavior that is prefixed with the sequence of system calls occurred before the invocation of the UB. We let $[\![prog]\!]_{\text{vRC11}} = \{ s \mid \mathsf{init}(prog) \Downarrow s \}$, which denotes the set of all behaviors that an initial machine state $\mathsf{init}(prog)$ of a program $prog$ exhibits.

Example 2. The "store buffering" test below demonstrates how the memory and the thread views of vRC11 captures weak behaviors exhibited by the reordering of a store followed by a load.

$$\begin{array}{l||l} X^{\texttt{rlx}} := 1 & Y^{\texttt{rlx}} := 1 \\ a := Y^{\texttt{rlx}} & b := X^{\texttt{rlx}} \\ \texttt{print}\ a & \texttt{print}\ b \end{array} \tag{SB}$$

Here, the behavior of both threads printing 0 is allowed by vRC11. Specifically, the first thread writes 1 to X by adding a message $\langle X@t, 1, \texttt{rlx}, \lambda X.\, 0 \rangle$ with some timestamp $t > 0$ and increasing its thread view of X to t. After the write, the thread reads from the initial message $\langle Y@0, 0, \texttt{na}, \lambda X.\, 0 \rangle$. By executing the second thread in the same way, it can read either from the initial message $\langle X@0, 0, \texttt{na}, \lambda X.\, 0 \rangle$ (since its view of X is still 0) or from the message of the first thread. Therefore, both threads can read 0 at the same execution.

Example 3. We show how vRC11 allows both threads printing 1 in the LB-NA example in §12. Suppose that the first thread reads 0 from the initial message for X, and writes 1 to Y by adding a message $m_Y = \langle Y@t, 1, \texttt{na}, \lambda X.\, 0 \rangle$ with some $t > 0$. Then, the read from Y by the second thread races with the message m_Y since it has a timestamp t greater than the timestamp of Y in the second thread's view (*i.e.*, $V(Y) = 0 < t$). Then, due to the racy read from Y, the second thread invokes UB, which generates arbitrary behavior, including the behavior in which both threads print 1.

Example 4. Consider the message passing program:

$$\begin{array}{l} D^{\mathtt{na}} := 42 \\ F^{\mathtt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := F^{\mathtt{acq}} \\ \mathtt{if}\ a = 1\ \mathtt{then} \\ \quad b := D^{\mathtt{na}} \end{array} \tag{MP}$$

The two non-atomic accesses to the data D are well-synchronized by a release-acquire synchronization through the flag F, and thus, they are not racy. Indeed, the first thread records its view in the message $F = 1$ and the view is transferred to the second thread when it reads $F = 1$. The read from D by the second thread is not racy since the timestamp of D it has in its view is already increased to include the timestamp of the message $D = 42$. Moreover, the second thread is only allowed to read 42 from D. In contrast, the program becomes racy if any (or both) of the accesses to F is made relaxed. Then, there would not be a release-acquire synchronization between the two threads, and the timestamp of D in the second thread's view would remain 0 (pointing to the initial message of D) even after reading 1 from F. In turn, the read from D would be racy and invoke a UB, as the message $D = 42$ would have a higher timestamp than the second thread's view of D.

We have ported the Coq proof by Cho et al. [18] to establish the *local DRF guarantees*, LDRF-RA and LDRF-SC, for vRC11. Generally speaking, data-race-freedom (DRF) guarantees ensure "strong" semantics for programs that are race-free under the "strong" semantics, and thus provide an essential formal justification for defensive programming. *Local* DRF (LDRF) guarantees further extend this idea to be applicable also in the presence of races on some unrelated locations (*e.g.*, confined in optimized libraries). LDRF-*RA* means that we consider release/acquire semantics as the strong semantics, and LDRF-*SC* means that under the strong semantics, threads can only access messages with globally maximal timestamps.

13.1 Relating vRC11 to RC11

The RC11 [37] memory model addresses two problems of the C/C++11 model: its flawed semantics for sequentially consistent accesses and fences (which is unrelated to the current paper) and the more crucial problem of "out-of-thin-air" reads [7] that breaks the fundamental DRF guarantee. To solve the latter problem, following [12],

RC11 takes a conservative approach and forbids cycles in the union of the program order and the reads-from relation. As discussed before, verification of concurrent programs under RC11 has been extensively studied and multiple verification methods and tools have been developed. The next theorem states that vRC11, the source model, is stronger than RC11. Hence, the soundness of all verification approaches for RC11 applies to vRC11 as well.

Theorem 10. For any program *prog*, $[\![prog]\!]_{\text{vRC11}} \subseteq [\![prog]\!]_{\text{RC11}}$.

We provide a (pen-and-paper) proof in §17.2, based on a declarative presentation of vRC11 (see §13.2), which can be more easily compared to RC11. Next, we demonstrate behaviors allowed by RC11 but disallowed by vRC11 using examples.

Putting presentation aside, the main difference between vRC11 and RC11 is related to the fact that an access in vRC11 can only race with previously executed writes, but not with previously executed reads. The following example ellustrates this point:

$$\begin{array}{l||l} a := X^{\mathtt{na}} & b := Y^{\mathtt{rlx}} \\ Y^{\mathtt{rlx}} := 1 & \mathtt{if}\ b = 1\ \mathtt{then}\ X^{\mathtt{na}} := 42 \end{array} \qquad \text{(RW-RACE)}$$

In both vRC11 and RC11, the read of X has to return 0, but this program is considered racy in RC11 but not in vRC11. Specifically, both vRC11 and RC11 allow the execution where the second thread reads 1 from Y (from the write of the first thread) and writes 42 to X. In RC11 this execution is deemed racy, since it has two accesses to the same location, such that (i) one of them is a write; (ii) one of them is non-atomic; and (iii) they are not properly synchronized by release/acquire accesses. In contrast, vRC11 does not view this execution as racy. In vRC11, an access can only race with a message to the same location that *already exists* in the memory. Thus, the write to X by the second thread is never racy since there has not been any other write to X. In other words, a write can never race with a read executed before the write. Note that the execution $X^{\mathtt{na}} := 42$ requires a message $Y = 1$ in the memory, so it cannot precede the read $a := X^{\mathtt{na}}$ by the first thread.

Remark 9. Deeming programs like RW-RACE as non-racy may allow performance improvements in certain programming idioms that are forbidden in RC11. For example,

consider the following multiple-readers-single-writer (MRSW) lock pattern:

$$\begin{array}{l} \ldots \\ a := X^{\mathtt{na}} \\ \texttt{reader-unlock}() \end{array} \;\Big\|\; \begin{array}{l} \ldots \\ b := X^{\mathtt{na}} \\ \texttt{reader-unlock}() \end{array} \;\Big\|\; \begin{array}{l} \texttt{writer-lock}() \\ X^{\mathtt{na}} := 42 \\ \ldots \end{array}$$

An MRSW lock protecting a location X allows multiple readers to read from X concurrently, while the writer should be exclusive, blocking any other reader or writer. A typical implementation of an MRSW lock maintains a counter counting how many readers currently hold the reader-lock. For such an implementation, `reader-unlock`() decreases the counter using a fetch-and-decrement operation and, `writer-lock`() checks if the counter reaches 0 and atomically swaps the value of the counter to some special value using a compare-and-swap. Under RC11, to prevent the race between the reads and the later write, `reader-unlock`() and `writer-lock`() should form release-acquire synchronization. In contrast, as in RW-RACE, such synchronization is unnecessary under vRC11 since a write never races with a read executed before the write. Therefore, vRC11 allows one to relax the write access mode of the fetch-and-add in `reader-unlock`() from `rel` to `rlx`. Moreover, when there is only one writer thread, `writer-lock`() can be further optimized to use a relaxed RMW instead of an acquire RMW.

In addition to the above, even for races with previously executed writes, the operational race condition of vRC11 is more restrictive than the race definition in RC11, where two accesses to the same location are considered racy if they are not "well-synchronized" (which is formally defined using the "happens-before" relation). This can be observed in programs when certain locations are accessed by both atomic and non-atomic accesses, as in the following example:

$$X^{\mathtt{rlx}} := 1 \;\Big\|\; \begin{array}{l} a := X^{\mathtt{rlx}} \\ \texttt{if } a = 1 \texttt{ then } b := X^{\mathtt{na}} \end{array} \qquad \text{(COH-RACE)}$$

In both vRC11 and RC11, the read of X has to return 1, but, again, this program is racy in RC11 but not in vRC11. To see this, consider an execution where the relaxed read $a := X^{\mathtt{rlx}}$ by the second thread reads 1 written by the first thread. The write $X^{\mathtt{rlx}} := 1$ by the first thread and the non-atomic read $b := X^{\mathtt{na}}$ by the second are racy

in RC11 since they are not well-synchronized via a release-acquire synchronization. In vRC11, the two accesses are not racy: once the second thread reads 1 by the relaxed read $a := X^{\texttt{rlx}}$, its view to X increases to include the message $X = 1$. Then, when the thread performs a non-atomic read from X, no message to X has a timestamp higher than the thread's view (*i.e.*, only the message $X = 1$ can be read by the second thread). Therefore, the non-atomic read by the second thread does not race with the write of the first thread.

Both of the above examples demonstrate cases that RC11 assigns UB to a program, whereas vRC11 gives it a defined semantics. We believe that races in vRC11 have a clear and simple meaning: a read access is racy iff it can read from more than one message, and a write access is racy iff it can "overwrite" more than one message. The examples above show cases where RC11 forces a non-atomic read to read from a particular write, but the read is still considered racy in RC11.

Finally, there also is a difference between the two models related to SC-fences (which are not presented above but included in the full model). In vRC11 the semantics of SC-fences is similar to the one in the RC20 model in [49], which is stronger than their semantics in RC11. In particular, SC-fences in vRC11 model can be expressed in terms of a release and acquire fences and an RMW to an otherwise unused location (see [49, Remark 1]). We do not discuss further this difference since it is orthogonal to our main topic.

13.2 A Declarative Presentation

For informed readers, we provide a declarative (a.k.a. axiomatic) presentation of vRC11. Such presentation is more concise than the operational one, and it is especially useful for comparing vRC11 to other models that are presented in a similar declarative fashion such as C/C++11. In the following, we consider the full model with RMWs and fences. Due to lack of space, we refer to [37], which we build on, for more background and examples of this definition style. (In any case, this technical section can be skipped when reading the paper.) The equivalence between the operational and declarative models is proved in §17.1.

In declarative models, program executions are represented by *execution graphs*, whose nodes, called *events*, keep track of accesses to the shared memory, and edges

provide several (partial) orders on these accesses. We assume that events are divided into three sets: writes ($\mathtt{W}$), reads ($\mathtt{R}$), and fences ($\mathtt{F}$). We use standard notations to retrieve events properties (such as $\mathtt{loc}(e)$ for the location accessed in e and $\mathtt{mod}(e)$ for the access mode) and to restrict sets accordingly (such as $\mathtt{W}^{\mathtt{rel}}$ for the set of release writes). Our execution graphs employ the standard basic relations: a program order ($\mathtt{po}$) that totally orders the events of each thread; an RMW relation ($\mathtt{rmw}$) that distinguishes the read-write pairs that together form an RMW; a reads-from relation ($\mathtt{rf}$) that links each write event w to the read events that read their value from w; and a modification order ($\mathtt{mo}$), a.k.a. coherence order, that totally orders all writes to the same location. Based on these relations, several other relations are derived (all as in RC11) using standard relational notations:

$$\mathtt{po}|_{\mathtt{loc}} \triangleq \{\langle e_1, e_2\rangle \in \mathtt{po} \mid \mathtt{loc}(e_1) = \mathtt{loc}(e_2)\} \qquad \textit{(po-same-location)}$$

$$\mathtt{rb} \triangleq \mathtt{rf}^{-1} ; \mathtt{mo} \qquad \textit{(reads-before, a.k.a. from-read)}$$

$$\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^{+} \qquad \textit{(extended-coherence-order)}$$

$$\mathtt{rs} \triangleq [\mathtt{W}] ; \mathtt{po}|^{?}_{\mathtt{loc}} ; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}] ; (\mathtt{rf} ; \mathtt{rmw})^{*} \qquad \textit{(release-sequence)}$$

$$\mathtt{sw} \triangleq ([\mathtt{W}^{\mathtt{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathtt{rel}}] ; \mathtt{po}) ; \mathtt{rs} ; \mathtt{rf} ; ([\mathtt{R}^{\mathtt{acq}}] \cup [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}] ; \mathtt{po} ; [\mathtt{F}^{\sqsupseteq \mathtt{acq}}]) \qquad \textit{(syncronized-with)}$$

$$\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^{+} \qquad \textit{(happens-before)}$$

Now, to handle SC-fences we include another primitive relation in execution graphs that determines the order of SC-fences. We call this relation the *SC-order*, denoted by $\mathtt{sc}$, and require it to be a *total* strict order on all the SC-fences (*i.e.*, on $\mathtt{F}^{\mathtt{sc}}$) in the execution graph. (Like $\mathtt{rf}$ and $\mathtt{mo}$, $\mathtt{sc}$ is existentially quantified—a behavior of a program is justified by *some* $\mathtt{sc}$ order of a corresponding graph.) Using $\mathtt{sc}$ we derive the *execution order*, which is a partial order on events that operational runs follow (note that $\mathtt{hb} \subseteq \mathtt{exec}$):

$$\mathtt{exec} \triangleq (\mathtt{po} \cup \mathtt{rf} \cup \mathtt{sc})^{+} \qquad \textit{(execution-order)}$$

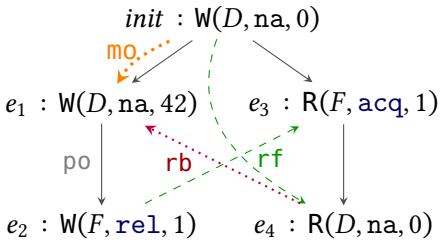
Then, consistent graphs are defined as follows.

Definition 1. An execution graph G is vRC11-*consistent* if the following hold for its relations:

- $\texttt{hb} ; \texttt{eco}$ is irreflexive. (COHERENCE)
- $\texttt{hb} ; \texttt{sc} ; \texttt{hb} ; \texttt{eco}$ is irreflexive. (SC-FENCE)
- $\texttt{rmw} \cap (\texttt{rb} ; \texttt{mo}) = \emptyset$. (ATOMICITY)
- $\texttt{exec}$ is irreflexive. (NO-LB)

The COHERENCE constraint is standard, and it ensures "SC-per-location". The SC-FENCE constraint gives the semantics to SC-fences, so they forbid, *e.g.*, store buffering behaviors when inserted between writes and reads. The ATOMICITY constraint ensures the atomicity of RMWs. Finally, NO-LB demonstrates that vRC11 is "in-order" as it entails the acyclicity of the union of the program order and the reads-from relation.

Example 5. As we saw in Example 4, in the MP program the second thread can only read 42 from D. To see how this follows from the declarative model, we depict the execution graph obtained when trying to read 0 (the initial value), and explain why it is inconsistent.



The nodes, labels, and program order ($\texttt{po}$) arise from the program behavior we analyze. Then, the reads-from relation ($\texttt{rf}$) is forced since every read has to read its value from some write writing that value. The modification order ($\texttt{mo}$) between *init* and e_1 is also forced: it has to order these two nodes (as both write to the same location), and going in the opposite order would violate COHERENCE as we have $\texttt{hb}$ from *init* to e_1. Then, according to the definition above, a "reads-before" edge ($\texttt{rb}$) is induced from e_4 to e_1, which implies $\langle e_4, e_1 \rangle \in \texttt{eco}$. Now, since e_2 and e_3 are $\texttt{rel}$ and $\texttt{acq}$, we have an $\texttt{sw}$ edge between them, inducing $\texttt{hb}$ from e_1 to e_4. Together, this violates COHERENCE since we have $\langle e_1, e_1 \rangle \in \texttt{hb} ; \texttt{eco}$.

To complete the presentation of the model, we define what execution graphs are considered *racy*, with the help of additional derived relations:

$$\texttt{conflict} \triangleq \left\{ \langle e_1, e_2 \rangle \;\middle|\; \begin{array}{l} e_1 \neq e_2 \wedge (\texttt{typ}(e_1) = \texttt{W} \vee \texttt{typ}(e_2) = \texttt{W}) \wedge \\ \texttt{loc}(e_1) = \texttt{loc}(e_2) \wedge (\texttt{mod}(e_1) = \texttt{na} \vee \texttt{mod}(e_2) = \texttt{na}) \end{array} \right\} \qquad \textit{(concliting events)}$$

$$\texttt{pb} \triangleq [\texttt{W}] ; \texttt{rf}^? ; \texttt{hb} ; \texttt{sc}^? ; \texttt{hb}^? \qquad \textit{(propagated-before)}$$

$$\texttt{raceWW} \triangleq [\texttt{W}] ; \texttt{conflict} ; [\texttt{W}] \setminus (\texttt{pb} \cup \texttt{exec}^{-1}) \qquad \textit{(write-write-race)}$$

$$\texttt{raceWR} \triangleq [\texttt{W}] ; \texttt{conflict} ; [\texttt{R}] \setminus (\texttt{pb} \cup \texttt{exec}^{-1}) \qquad \textit{(write-read-race)}$$

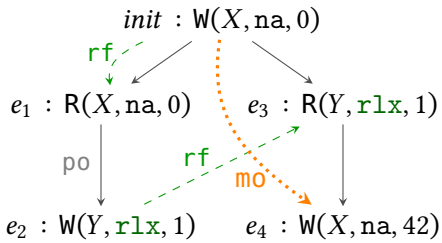
Roughly, $\langle w, e \rangle \in \texttt{pb}$ means that the write w has been observed by the thread executing e before it executes e, where observations are propagated through release/acquire synchronization and SC-fences. In the operational model, this means that (the message associated with) w has a timestamp lower than the timestamp of the location of w in the current view of the thread executing e. Then, raceWW relates two conflicting writes, w_1 to w_2, when w_1 has not propagated before w_2 ($\langle w_1, w_2 \rangle \notin \texttt{pb}$) and w_1 can be executed before w_2 ($\langle w_2, w_1 \rangle \notin \texttt{exec}$). Similarly, raceWR relates conflicting write and read, w to r, when w has not propagated before r ($\langle w, r \rangle \notin \texttt{pb}$) and w can be executed before r ($\langle r, w \rangle \notin \texttt{exec}$).

Finally, we say that execution graph G is vRC11-*racy* if $\texttt{raceWW} \cup \texttt{raceWR} \neq \emptyset$, and as in RC11, a program outcome is allowed if it is induced by some vRC11-consistent execution graph that is generated by the program, or if *some* racy vRC11-consistent execution graph is generated by the program. The latter disjunct corresponds to the program invoking UB.

Remark 10. An equivalent model is obtained if we include sc inside hb (together with po and sw). In this presentation, SC-FENCE is not needed, and the definition of pb can be simplified to $\texttt{rf}^?$; hb.

Example 6. The RW-RACE program above does not generate a racy vRC11-consistent execution graph. Indeed, the only vRC11-consistent execution graph of it executing

both non-atomic accesses is depicted below.



Here, we have $\langle e_4, e_1\rangle \in [\mathtt{W}]; \mathtt{conflict}; [\mathtt{R}] \setminus \mathtt{pb}$, but this is *not* considered a write-read race since $\langle e_1, e_4\rangle \in \mathtt{exec}$. In turn, if we had $X^{\mathtt{na}} := 1$ in the first thread (rather than $a := X^{\mathtt{na}}$), we would obtain that $\langle e_1, e_4\rangle \in \mathtt{raceWW}$ (with e_1 labeled by $\mathtt{W}(X, \mathtt{na}, 1)$) which would mean that the program has a racy vRC11-consistent execution graph, so it allows any outcome.

14 The IR Model

In this section, we present our model of the intermediate representation, called $\mathrm{PS}^{\mathtt{IR}}$, establish the soundness of mapping from vRC11 to $\mathrm{PS}^{\mathtt{IR}}$, and show that every sound transformation under the sequential semantics SEQ in [18] is also sound under $\mathrm{PS}^{\mathtt{IR}}$.

There are two major differences between $\mathrm{PS}^{\mathtt{IR}}$ and vRC11. First, to allow load introduction, a racy read in $\mathrm{PS}^{\mathtt{IR}}$ returns an *undefined value* instead of invoking UB. As demonstrated above, this change alone forbids the weak behavior of LB-NA. $\mathrm{PS}^{\mathtt{IR}}$ addresses this issue by allowing *promises*, so it is not an in-order semantics. Intuitively, a thread may promise that it will perform a non-atomic write to a location X in the future, making any accesses to X by other threads to be racy.

In the following, we explain the $\mathrm{PS}^{\mathtt{IR}}$ model focusing on how it extends and modifies vRC11. Figure IV.5 summarizes the steps of $\mathrm{PS}^{\mathtt{IR}}$. We note that the (SILENT) and (READ) steps are adapted in an obvious way that does not alter the new components of the state, and the (RACE) definition is also exactly as in vRC11. The full $\mathrm{PS}^{\mathtt{IR}}$ model in our Coq development includes RMWs, fences, and release sequences, and is presented in §15.2.

Promises Both each thread state and the global machine state are equipped with a set of locations, called *local promises* (P) and *global promises* (P_G), respectively. The

$P, P_G \subseteq \mathsf{Loc}$	promise set
$T = \langle \sigma, V, P \rangle \in \mathsf{Lts}$	thread state
$\langle T, P_G, M \rangle$	thread configuration
$\langle \mathcal{T}, P_G, M \rangle$	machine state

(SILENT)
$$\frac{\sigma \to \sigma'}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \sigma', V, P \rangle, P_G, M\rangle}$$

(PROMISE)
$$\frac{X \notin P_G}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \sigma, V, P \cup \{X\} \rangle, P_G \cup \{X\}, M\rangle}$$

(SYSTEM CALL)
$$\frac{\sigma \xrightarrow{\mathsf{Sys}(e)} \sigma' \qquad P = \emptyset}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \xrightarrow{\mathsf{Sys}(e)} \langle\langle \sigma', V, P \rangle, P_G, M\rangle}$$

(READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, v)} \sigma' \qquad \langle X@t, v, \_, V_\mathsf{m}\rangle \in M \qquad V(X) \le t \qquad V' = V[X \mapsto t] \sqcup \begin{cases} 0 & o_\mathsf{R} \neq \mathtt{acq} \\ V_\mathsf{m} & o_\mathsf{R} = \mathtt{acq} \end{cases}}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \sigma', V', P \rangle, P_G, M\rangle}$$

(WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, v)} \sigma' \qquad m = \langle X@t, v, o_\mathsf{W}, V_\mathsf{m}\rangle \qquad M \# m \qquad V(X) < t \qquad V' = V[X \mapsto t] \qquad V_\mathsf{m} = \begin{cases} \lambda X.\, 0 & o_\mathsf{W} \neq \mathtt{rel} \\ V' & o_\mathsf{W} = \mathtt{rel} \end{cases} \qquad \langle P', P_G{}' \rangle = \begin{cases} \langle P \setminus \{X\}, P_G \setminus \{X\} \rangle & o_\mathsf{W} = \mathtt{na} \wedge X \in P \\ \langle P, P_G \rangle & \text{otherwise} \end{cases}}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \sigma', V', P' \rangle, P_G{}', M \cup \{m\}\rangle}$$

(RACE)
$$\frac{\langle X@t, \_, o_\mathsf{W}, \_\rangle \in M \qquad V(X) < t \qquad o_\mathsf{W} = \mathtt{na} \vee o = \mathtt{na}}{\mathsf{race}(V, M, X, o)}$$

(PROMISED RACE)
$$\frac{X \in P_G \setminus P}{\mathsf{race}_\mathsf{prm}(P, P_G, X)}$$

(RACY-READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, \mathtt{undef})} \sigma' \qquad \mathsf{race}(V, M, X, o_\mathsf{R}) \vee \mathsf{race}_\mathsf{prm}(P, P_G, X)}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \sigma', V, P \rangle, P_G, M\rangle}$$

(RACY-WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, \_)} \_ \qquad \mathsf{race}(V, M, X, o_\mathsf{W})}{\langle\langle \sigma, V, P \rangle, P_G, M\rangle \to \langle\langle \bot, V, \emptyset \rangle, P_G, M\rangle}$$

(MACHINE: NORMAL)
$$\frac{\langle \mathcal{T}(\tau), P_G, M\rangle \to^+ \langle T', P_G{}', M'\rangle \qquad \langle T', P_G{}', M'\rangle \to^* \langle\langle \_, \_, \emptyset \rangle, \_, \_\rangle}{\langle \mathcal{T}, P_G, M\rangle \to \langle \mathcal{T}[\tau \mapsto T'], P_G{}', M'\rangle}$$

(MACHINE: UB)
$$\frac{\langle \mathcal{T}(\tau), P_G, M\rangle \to^+ \langle\langle \bot, \_, \_\rangle, P_G{}', M'\rangle}{\langle \mathcal{T}, P_G, M\rangle \to \langle \bot, P_G{}', M'\rangle}$$

Figure IV.5: Domains and transitions of $\mathrm{PS}^{\mathtt{IR}}$ (RMWs, fences, release sequences, and reservations are omitted).

(PROMISE) step allows a thread to *promise* to write to a certain location X in the future by adding X both to its local promises and to the global promises, provided that X has not been already promised by some thread ($X \notin P_G$). Once X being promised, the thread can later *fulfill* its promise by writing to X via a non-atomic write. Accordingly, the (WRITE) step is extended to update the local and global promises set by removing X from them if the write is non-atomic and X was previously promised.

Certification At each machine step (see (MACHINE: NORMAL)), the thread taking a sequence of steps should *certify* its promises by demonstrating it is able to fulfill all its promises by taking multiple steps in isolation. The certification requirement is crucial in proving the soundness of mapping from vRC11 to PS^{IR} (see Example 9 below).

Racy Reads A racy read retrieves an undefined value (denoted by `undef` in (RACY-READ)) (unlike invoking UB in vRC11), thereby allowing the compiler transformation that introduces unused loads. In addition, a race occurs with promised writes (see (PROMISED RACE) definition): a memory access to X is considered racy also if there is a promise to X made by *another* thread.

Racy Writes A racy write invokes UB (like in vRC11), and there is no need to consider promised writes for these races. A thread transition invoking UB directly fulfills the remaining promises (see (RACY-WRITE)), so the local promises set is made empty after the transition, which allows for a successful certification process.

System Calls A system call requires the local promises to be empty ($P = \emptyset$). In other words, a write cannot be promised over a system call. Intuitively, this means that a system call followed by a (non-atomic) write cannot be reordered.

Behavior Program behaviors under PS^{IR} are defined as for vRC11, with one modification: when `undef` is a part of a system call output in an execution trace, then it can be refined to any concrete value in the program behavior (*e.g.*, `print(undef)` in a trace can be mapped to `print(1)` in the behavior). We denote by $[\![prog]\!]_{\text{PS}^{\text{IR}}}$ the set of behaviors a program *prog* exhibits under PS^{IR}.

Example 7. In contrast to vRC11, in which a racy read invokes UB, in PS$^{\mathtt{IR}}$ a racy read returns `undef`. Thus, PS$^{\mathtt{IR}}$ justifies LB-NA example using a promise. Specifically, the first thread promises to Y, certified by reading 0 from X and writing 1 to Y. Then, the read from Y by the second thread is racy with that promise and it returns `undef`. After the racy read, the second thread writes 1 to X and prints 1. (Since `undef` represents an arbitrary value, it can be refined to 1.) Now, the first thread reads 1 from X, fulfills its promise to Y by performing a non-atomic write, and prints 1 as well.

Example 8. The following example demonstrates that a promise can be certified and fulfilled by different write instructions even with different written values.

$$\begin{array}{l} a := X^{\mathtt{na}} \\ Y^{\mathtt{rlx}} := a \end{array} \;\Bigg\|\; \begin{array}{l} b := Y^{\mathtt{rlx}} \\ \mathtt{if}\ b \neq 0\ \mathtt{then} \\ \quad X^{\mathtt{na}} := 1 \\ \quad \mathtt{print}\ 42 \\ \mathtt{else} \\ \quad X^{\mathtt{na}} := 2 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := X^{\mathtt{na}} \\ Y^{\mathtt{rlx}} := a \end{array} \;\Bigg\|\; \begin{array}{l} X^{\mathtt{na}} := 2 \\ b := Y^{\mathtt{rlx}} \\ \mathtt{if}\ b \neq 0\ \mathtt{then} \\ \quad X^{\mathtt{na}} := 1 \\ \quad \mathtt{print}\ 42 \end{array} \qquad \text{(LB-CASE)}$$

The program on the left can be transformed into the program on the right by applying a sequence of compiler transformations in the second thread: (*i*) split the non-atomic write $X^{\mathtt{na}} := 1$ into two writes $X^{\mathtt{na}} := 2$ followed by $X^{\mathtt{na}} := 1$; (*ii*) hoist the common write $X^{\mathtt{na}} := 2$ out of the branch; and (*iii*) reorder the read $b := Y^{\mathtt{rlx}}$ and the write $X^{\mathtt{na}} := 2$. Since the program on the right is allowed to print 42 (even under SC), PS$^{\mathtt{IR}}$ should allow the same behavior for the program on the left. Indeed, the program on the left can print 42 through the following PS$^{\mathtt{IR}}$ execution: (*i*) the second thread promises to X, certifying it by reading 0 from Y and writing 2 to X; (*ii*) the first thread reads `undef` from X by performing a racy read and writes `undef` to Y; and (*iii*) the second thread reads `undef` from Y, enters the then-branch as $b \neq 0$ evaluates to `undef`,[12] fulfills its promise to X by writing 1 to X, and prints 42. Notably, in this execution, the promise to X of the second thread is certified using the write $X^{\mathtt{na}} := 2$ and fulfilled by the other write $X^{\mathtt{na}} := 1$.

[12]Here, we assume that branching on `undef` non-deterministically takes either one of the then-branch or the else-branch. In LLVM, branching on `undef` is UB, and a "freeze" instruction should be used before the branching [40]. `freeze`(v) returns v when v is a defined value (*i.e.*, not `undef`) and non-deterministically returns any defined value (*e.g.*, 42) when v is `undef`. In the example, the same argument holds when $b \neq 0$ is replaced with `freeze`(b) $\neq 0$.

Our main result is the following theorem (a Coq proof is available in the supplementary material):

Theorem 11. For any program $prog$, $[\![prog]\!]_{\mathrm{PS}^{\mathtt{IR}}} \subseteq [\![prog]\!]_{\mathrm{vRC11}}$.

The proof of this theorem is established using a simulation argument: for each transition in $\mathrm{PS}^{\mathtt{IR}}$, we identify a corresponding sequence of transitions in vRC11. While most thread transitions are identical in vRC11 and $\mathrm{PS}^{\mathtt{IR}}$, there are no corresponding transitions in vRC11 for the following two transitions of $\mathrm{PS}^{\mathtt{IR}}$: 1. (PROMISE) transition; and 2. (RACY-READ) transition that races with a promise (*i.e.*, when $\mathsf{race}_{\mathsf{prm}}(P, P_G, X)$ holds). For the former, the vRC11 machine simply takes no transition and the machine states of vRC11 and $\mathrm{PS}^{\mathtt{IR}}$ remains identical except for the sets of promises (P and P_G) in $\mathrm{PS}^{\mathtt{IR}}$. For the latter, we show that the vRC11 machine can take multiple steps and invoke UB by performing a racy read. Concretely, suppose that a thread τ_1 of $\mathrm{PS}^{\mathtt{IR}}$ performs a racy read from a location X that races with a promise made by another thread τ_2. Since there is no promise in vRC11, we need to prove that τ_2 of vRC11 can actually perform a non-atomic write to X before τ_1 takes a racy read transition. The key property in this is to turn a certification of the promise by τ_2 in $\mathrm{PS}^{\mathtt{IR}}$ into a real execution of vRC11. To do so, we proved that once a thread certifies its promise to a location X, under any possible future memory, the thread can take multiple steps and perform a non-atomic write to X.

Example 9. Theorem 11 does not hold without the certification of promises.

$$\begin{array}{l||l} a := X^{\mathtt{na}} & b := Y^{\mathtt{na}} \\ \mathtt{if}\ a = 1\ \mathtt{then} & \mathtt{if}\ b = 1\ \mathtt{then} \\ \quad Y^{\mathtt{na}} := 1 & \quad X^{\mathtt{na}} := 1 \end{array} \tag{LB-DRF}$$

Under vRC11, the only possible execution for this program is that both threads read the initial messages (*i.e.*, $a = b = 0$). For Thm. 11 to hold, $\mathrm{PS}^{\mathtt{IR}}$ cannot allow any other behavior. Indeed, under $\mathrm{PS}^{\mathtt{IR}}$, the first thread cannot promise to Y since the only value that can be read from X is 0, and thus, the thread cannot *certify* the promise. However, if $\mathrm{PS}^{\mathtt{IR}}$ allowed a thread to promise without certifying it, then $a = b = \mathtt{undef}$ would be allowed. Specifically, the first thread could unconditionally promise to Y; the second thread could read `undef` and write 1 to X; and then the first thread could read `undef` and write 1 to Y while fulfilling its promise to Y.

$P \subseteq \mathsf{Loc}$ permission set | $M \in \mathsf{Loc} \to \mathsf{Val}$ sequential memory

$F \subseteq \mathsf{Loc}$ written locations set | $S = \langle \sigma, P, F, M \rangle$ thread state of SEQ

(SILENT)
$$\frac{\sigma \to \sigma'}{\langle \sigma, P, F, M \rangle \to \langle \sigma', P, F, M \rangle}$$

(CHOICE/RELAXED-READ)
$$\frac{\sigma \xrightarrow{e} \sigma' \qquad e \in \{\mathtt{choose}(v), \mathsf{R}^{\mathtt{rlx}}(x, v)\}}{\langle \sigma, P, F, M \rangle \xrightarrow{e} \langle \sigma', P, F, M \rangle}$$

(NA-READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}^{\mathtt{na}}(X,v)} \sigma' \qquad X \in P \qquad v = M(X)}{\langle \sigma, P, F, M \rangle \to \langle \sigma', P, F, M \rangle}$$

(NA-WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}^{\mathtt{na}}(X,v)} \sigma' \qquad X \in P \qquad F' = F \cup \{X\} \qquad M' = M[X \mapsto v]}{\langle \sigma, P, F, M \rangle \to \langle \sigma', P, F', M' \rangle}$$

(RACY-NA-READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}^{\mathtt{na}}(X,\mathtt{undef})} \sigma' \qquad X \notin P}{\langle \sigma, P, F, M \rangle \to \langle \sigma', P, F, M \rangle}$$

(RACY-NA-WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}^{\mathtt{na}}(X,\_)} \_ \qquad X \notin P}{\langle \sigma, P, F, M \rangle \to \langle \bot, P, F, M \rangle}$$

(ACQ-READ)
$$\frac{\sigma \xrightarrow{\mathsf{R}^{\mathtt{acq}}(x,v)} \sigma' \qquad P \subseteq P' \qquad dom(V) = P' \setminus P \qquad M' = \lambda X. \begin{cases} V(X) & X \in P' \setminus P \\ M(X) & \text{otherwise} \end{cases}}{\langle \sigma, P, F, M \rangle \xrightarrow{\mathsf{R}^{\mathtt{acq}}(x,v,P,P',F,V)} \langle \sigma', P', F, M' \rangle}$$

(RELAXED-WRITE/REL-WRITE)
$$\frac{\sigma \xrightarrow{\mathsf{W}^{\mathtt{rel}}(x,v)} \sigma' \qquad P' \subseteq P \qquad V = M|_P}{\langle \sigma, P, F, M \rangle \xrightarrow{\mathsf{W}^{\mathtt{rel}}(x,v,P,P',F,V)} \langle \sigma', P', \emptyset, M \rangle}$$

Figure IV.6: Transitions of SEQ adapted for $\mathrm{PS}^{\mathtt{IR}}$. Compared to SEQ by Cho et al. [18], the rule for relaxed writes is strengthened to share the one for release writes (highlighted in the figure). Consequently, SEQ here forbids reordering of a non-atomic access followed by a relaxed write.

In addition to Thm. 11, we also proved that program transformations sound in sequential semantics are also sound to apply on non-atomics in $\mathrm{PS}^{\mathtt{IR}}$. To do so, we adapted the sequential machine SEQ from [18, Def. 3.3] to include non-promisable relaxed writes, as we have in $\mathrm{PS}^{\mathtt{IR}}$, and ported the proof in [18] to $\mathrm{PS}^{\mathtt{IR}}$ to show that all sound optimizations on non-atomics under (the adapted) SEQ are also sound under $\mathrm{PS}^{\mathtt{IR}}$. The transition rules for SEQ are given in Fig. IV.6. Thanks to this result, not only the programmers but also compiler writers who develop optimizations on non-atomic code (including reorderings and eliminations of non-atomic accesses across atomics) do not have to understand the out-of-order IR model.

The sequential machine SEQ, however, is not helpful for validating reorderings

and eliminations of *atomics*. These optimizations are not important for our current purpose (to the best of our knowledge, they are not performed by current compilers). In fact, we found out that reordering of relaxed writes (to different locations) is unsound in PS^{IR}. (Still, PS^{IR} can be soundly mapped to Armv8, which effectively allows reordering in the target code; see §16.) The reason is related to the *reservation* mechanism, an addition that was introduced to PS in [44] and used in our full PS^{IR} model, in order to support an efficient mapping of RMWs to Armv8. Future work is required to understand whether PS^{IR} can be changed to allow this reordering. We expect all other transformations on atomics that are sound in RC11 to be sound in PS^{IR}.

15 Full models

In this section, we present the full models of vRC11 and PS^{IR}.

15.1 The Full vRC11 Model

We first present the full vRC11 model including RMWs, fences, and release sequences. In the following, we focus on the components that are not introduced in §13. Figure IV.7 provides the domains and helper rules for vRC11, where the domains are identical to Fig. IV.4 if not listed here. Note that we assume read access modes are ordered by $\texttt{na} \sqsubseteq \texttt{rlx} \sqsubseteq \texttt{acq}$, and write access modes are ordered by $\texttt{na} \sqsubseteq \texttt{rlx} \sqsubseteq \texttt{rel}$.

Memory A *memory* is a (nonempty) pairwise disjoint finite set of messages. Now, a *message* $\langle X@(f,t], v, o_{\mathsf{W}}, V_{\mathsf{m}} \rangle \in \mathsf{Msg}$ carries a timestamp interval $(f,t]$, where, $f < t$ instead of a single timestamp. We denote by $m.\texttt{loc}$, $m.\texttt{val}$, $m.\texttt{from}$, $m.\texttt{to}$, $m.\texttt{mod}$, and $m.\texttt{view}$ the components of a message m. We write two messages m_1 and m_2 are disjoint, denoted by $m_1 \# m_2$, if they have different locations ($m_1.\texttt{loc} \neq m_2.\texttt{loc}$) or have disjoint timestamp intervals ($m_1.\texttt{to} < m_2.\texttt{from} \vee m_2.\texttt{to} < m_1.\texttt{from}$). A memory M and a message m are disjoint (denoted by $M \# m$) if $\forall m' \in M.\ m' \# m$.

States A *machine state* $\langle \mathcal{T}, S, M \rangle$ now includes an SC-view $S \in V$. A *thread state* is a tuple $T = \langle \sigma, \mathcal{V} \rangle$, where σ is a local program state as before, and $\mathcal{V}$ is a thread

$V_{\mathtt{rel}} \in \mathsf{Loc} \to \mathsf{View}$	release view	$m = \langle X@(f,t], v, o_{\mathtt{W}}, V\rangle \in \mathsf{Msg}$	message
$V_{\mathtt{cur}} \in \mathsf{View}$	current view	$o_{\mathtt{F}} \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$	fence access mode
$V_{\mathtt{acq}} \in \mathsf{View}$	acquire view	$T = \langle \sigma, \mathcal{V}\rangle \in \mathsf{Lts}$	thread state
$\mathcal{V} = \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle$	thread view	$\langle T, S, M\rangle$	thread configuration
$S \in \mathsf{View}$	sc view	$\langle \mathcal{T}, S, M\rangle$	machine state

(READ-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(\_,t], \_, \_, V_{\mathtt{m}}\rangle \\ V_{\mathtt{cur}}(X) \le t \qquad V_s = [X \mapsto t] \\ V'_{\mathtt{cur}} = V_{\mathtt{cur}} \sqcup V_s \sqcup (o_{\mathtt{R}} \sqsupseteq \mathtt{acq}\,?\,V_{\mathtt{m}}) \\ V'_{\mathtt{acq}} = V_{\mathtt{acq}} \sqcup V_s \sqcup (o_{\mathtt{R}} \sqsupseteq \mathtt{rlx}\,?\,V_{\mathtt{m}}) \end{array}}{\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle \xrightarrow{o_{\mathtt{R}}, m}_{\mathtt{R}} \langle V_{\mathtt{rel}}, V'_{\mathtt{cur}}, V'_{\mathtt{acq}}\rangle}$$

(WRITE-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(f,t], \_, \_, V_{\mathtt{m}}\rangle \qquad f < t \\ V_{\mathtt{cur}}(X) < t \qquad V_s = [X \mapsto t] \\ V'_{\mathtt{cur}} = V_{\mathtt{cur}} \sqcup V_s \qquad V'_{\mathtt{acq}} = V_{\mathtt{acq}} \sqcup V_s \\ V'_{\mathtt{rel}} = V_{\mathtt{rel}}[X \mapsto V_{\mathtt{rel}}(X) \sqcup V_s \sqcup (o_{\mathtt{W}} \sqsupseteq \mathtt{rel}\,?\,V'_{\mathtt{cur}})] \\ V_{\mathtt{m}} = (o_{\mathtt{W}} \sqsupseteq \mathtt{rlx}\,?\,(V'_{\mathtt{rel}}(X) \sqcup V_r)) \end{array}}{\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle \xrightarrow{o_{\mathtt{W}}, V_r, m}_{\mathtt{W}} \langle V'_{\mathtt{rel}}, V'_{\mathtt{cur}}, V'_{\mathtt{acq}}\rangle}$$

(RACE-HELPER)

$$\frac{\begin{array}{c} \langle X@(\_,t], \_, o_{\mathtt{W}}, \_\rangle \in M \\ \mathcal{V}.\mathtt{cur}(X) < t \\ o_{\mathtt{W}} = \mathtt{na} \vee o = \mathtt{na} \end{array}}{\mathsf{race}(\mathcal{V}, M, X, o)}$$

(FENCE-HELPER: NON-SC)

$$\frac{\mathcal{V}' = \begin{cases} \langle V_{\mathtt{rel}}, V_{\mathtt{acq}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{acq} \\ \langle \lambda\_.\, V_{\mathtt{cur}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{rel} \\ \langle \lambda\_.\, V_{\mathtt{acq}}, V_{\mathtt{acq}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{acqrel} \end{cases}}{\langle\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle, S\rangle \xrightarrow{o_{\mathtt{F}}}_{\mathtt{F}} \langle \mathcal{V}', S\rangle}$$

(FENCE-HELPER: SC)

$$\frac{\begin{array}{c} S' = \mathcal{V}.\mathtt{acq} \sqcup S \\ \mathcal{V}' = \langle \lambda\_.\, S', S', S'\rangle \end{array}}{\langle \mathcal{V}, S\rangle \xrightarrow{\mathtt{sc}}_{\mathtt{F}} \langle \mathcal{V}', S'\rangle}$$

Figure IV.7: The full vRC11 model (domains and auxiliary definitions).

view. A *thread view* is a triple $\mathcal{V} = \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle$, where $V_{\mathtt{rel}} \in \mathsf{Loc} \to \mathsf{View}$ and $V_{\mathtt{cur}}, V_{\mathtt{acq}} \in \mathsf{View}$. We denote by $\mathcal{V}.\mathtt{cur}$, $\mathcal{V}.\mathtt{acq}$, and $\mathcal{V}.\mathtt{rel}$ the components of $\mathcal{V}$. Note that $V_{\mathtt{rel}}(X) \sqsubseteq V_{\mathtt{cur}} \sqsubseteq V_{\mathtt{acq}}$ always holds for any X throughout the execution.

Thread Configuration Steps Thread configuration steps are defined in Fig. IV.8. SILENT, READ, WRITE, and RACY-READ/WRITE steps are naturally extended from the simple model presented in Fig. IV.4. In the following, we describe the remaining thread transition rules.

UPDATE. The UPDATE step first reads a message with a timestamp interval $(\_, t]$ and writes a new message with an interval $(t, \_]$, attaching it to the read message.

RACY-UPDATE. Since every RMW operation is atomic, we simply exclude RMWs with a non-atomic access mode by considering them to be racy. Then, similarly to RACY-WRITE step, an atomic RMW with both access modes higher than or equal

(SILENT)

$$\frac{\sigma \to \sigma'}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}\rangle, S, M\rangle}$$

(READ)

$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, v)} \sigma' \qquad m = \langle X@(\_, \_], v, \_, \_\rangle \in M \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m}_\mathsf{R} \mathcal{V}'}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}'\rangle, S, M\rangle}$$

(WRITE)

$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, v)} \sigma' \qquad m = \langle X@(\_, \_], v, \_, \_\rangle \qquad M \# m \qquad \mathcal{V} \xrightarrow{o_\mathsf{W}, \lambda\_.0, m}_\mathsf{W} \mathcal{V}' \qquad M' = M \cup \{m\}}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}'\rangle, S, M'\rangle}$$

(UPDATE)

$$\frac{\sigma \xrightarrow{\mathsf{RMW}(X, o_\mathsf{R}, o_\mathsf{W}, v_\mathsf{R}, v_\mathsf{W})} \sigma' \qquad m_\mathsf{R} = \langle X@(\_, t_\mathsf{R}], v_\mathsf{R}, \_, V_r\rangle \in M \qquad m_\mathsf{W} = \langle X@(t_\mathsf{R}, t_\mathsf{W}], v_\mathsf{W}, \_, \_\rangle \qquad M \# m_\mathsf{W} \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m_\mathsf{R}}_\mathsf{R} \xrightarrow{o_\mathsf{W}, V_r, m_\mathsf{W}}_\mathsf{W} \mathcal{V}' \qquad M' = M \cup \{m_\mathsf{W}\}}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}'\rangle, S, M'\rangle}$$

(RACY-READ)

$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, \_)} \_ \qquad \mathsf{race}(\mathcal{V}, M, X, o_\mathsf{R})}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \bot, \mathcal{V}\rangle, S, M\rangle}$$

(RACY-WRITE)

$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, \_)} \_ \qquad \mathsf{race}(\mathcal{V}, M, X, o_\mathsf{W})}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \bot, \mathcal{V}\rangle, S, M\rangle}$$

(RACY-UPDATE)

$$\frac{\sigma \xrightarrow{\mathsf{RMW}(X, o_\mathsf{R}, o_\mathsf{W}, \_, \_)} \_ \qquad o_\mathsf{R} = \mathtt{na} \vee o_\mathsf{W} = \mathtt{na} \vee \mathsf{race}(\mathcal{V}, M, X, \mathtt{rlx})}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \bot, \mathcal{V}\rangle, S, M\rangle}$$

(FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F}(o_\mathsf{F})} \sigma' \qquad \langle \mathcal{V}, S\rangle \xrightarrow{o_\mathsf{F}}_\mathsf{F} \langle \mathcal{V}', S'\rangle}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \to \langle\langle \sigma', \mathcal{V}'\rangle, S', M\rangle}$$

(SYSTEM CALL)

$$\frac{\sigma \xrightarrow{\mathsf{Sys}(e)} \sigma' \qquad \langle \mathcal{V}, S\rangle \xrightarrow{\mathtt{sc}}_\mathsf{F} \langle \mathcal{V}', S'\rangle}{\langle\langle \sigma, \mathcal{V}\rangle, S, M\rangle \xrightarrow{\mathsf{Sys}(e)} \langle\langle \sigma', \mathcal{V}'\rangle, S', M\rangle}$$

Figure IV.8: The full vRC11 model (thread transitions).

$$\frac{\langle \mathcal{T}(\tau), S, M\rangle \xrightarrow{l} \langle T', S', M'\rangle}{\langle \mathcal{T}, S, M\rangle \xrightarrow{l} \langle \mathcal{T}[\tau \mapsto T'], S', M'\rangle}\ (\textsc{machine: normal}) \qquad \frac{\langle \mathcal{T}(\tau), S, M\rangle \to \langle\langle \bot, \_\rangle, S', M'\rangle}{\langle \mathcal{T}, S, M\rangle \to \langle \bot, S', M'\rangle}\ (\textsc{machine: ub})$$

Figure IV.9: The full vRC11 model (machine transitions).

`rlx` is racy if there is a non-atomic message in the location being accessed with the to-timestamp higher than the thread's current view.

FENCE. A FENCE step updates the thread's view following FENCE-HELPER: NON-SC rule. If a thread executes a sequentially consistent fence, it increases all its views and SC timemap by incorporating the thread's acquire view and the SC timemap before taking the step.

SYSTEM CALL. A SYSTEM CALL step is similar to the one in Fig. IV.4 except that the rule here additionally executes an SC fence.

Machine Steps Figure IV.9 presents the machine steps, which are defined just like in the simple model presented in Fig. IV.4.

15.2 The Full PS$^{\mathtt{IR}}$ Model

Now, we present the full PS$^{\mathtt{IR}}$ model. Compared to the simplified version in §14, the full model includes RMWs, fences, release sequences, and reservation mechanism. As before, the following description focuses on these additional notions.

Reservation A *reservation* $r = \langle X@(f, t]\rangle \in \mathsf{Rsv}$ is a valueless message. A thread can reserve a timestamp interval of a certain location (RESERVE rule in Fig. IV.11) in order to prevent other threads from writing to that interval. Later the thread may cancel its reservation (CANCEL rule in Fig. IV.11) and write to the reserved space. A memory is naturally extended to be a set of messages and reservations.

States Similarly to the full vRC11 model, a *machine state* $\langle \mathcal{T}, P_G, S, M\rangle$ includes an SC-view $S \in V$. A *thread state* is a tuple $T = \langle \sigma, \mathcal{V}, P, R\rangle$, where R is a set of reservations, and other components are the same as in the full vRC11 model.

$V_{\mathtt{rel}} \in \mathsf{Loc} \to \mathsf{View}$	release view	$m = \langle X@(f,t], v, o_{\mathtt{W}}, V\rangle \in \mathsf{Msg}$	message
$V_{\mathtt{cur}} \in \mathsf{View}$	current view	$R \subseteq \mathsf{Rsv}$	reserve set
$V_{\mathtt{acq}} \in \mathsf{View}$	acquire view	$M \subseteq \mathsf{Msg} \cup \mathsf{Rsv}$	memory
$\mathcal{V} = \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle$	thread view	$o_{\mathtt{F}} \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$	fence access mode
$S \in \mathsf{View}$	sc view	$T = \langle \sigma, \mathcal{V}, P, R\rangle \in \mathsf{Lts}$	thread state
$P, P_G \subseteq \mathsf{Loc}$	promises set	$\langle T, P_G, S, M\rangle$	thread configuration
$r = \langle X@(f,t]\rangle \in \mathsf{Rsv}$	reserve	$\langle \mathcal{T}, P_G, S, M\rangle$	machine state

(READ-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(\_,t], \_, \_, V_{\mathtt{m}}\rangle \\ V_{\mathtt{cur}}(X) \le t \qquad V_s = [X \mapsto t] \\ V'_{\mathtt{cur}} = V_{\mathtt{cur}} \sqcup V_s \sqcup (o_{\mathtt{R}} \sqsupseteq \mathtt{acq}\,?\,V_{\mathtt{m}}) \\ V'_{\mathtt{acq}} = V_{\mathtt{acq}} \sqcup V_s \sqcup (o_{\mathtt{R}} \sqsupseteq \mathtt{rlx}\,?\,V_{\mathtt{m}}) \end{array}}{\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle \xrightarrow{o_{\mathtt{R}}, m}_{\mathtt{R}} \langle V_{\mathtt{rel}}, V'_{\mathtt{cur}}, V'_{\mathtt{acq}}\rangle}$$

(WRITE-HELPER)

$$\frac{\begin{array}{c} m = \langle X@(f,t], \_, \_, V_{\mathtt{m}}\rangle \qquad f < t \\ V_{\mathtt{cur}}(X) < t \qquad V_s = [X \mapsto t] \\ V'_{\mathtt{cur}} = V_{\mathtt{cur}} \sqcup V_s \qquad V'_{\mathtt{acq}} = V_{\mathtt{acq}} \sqcup V_s \\ V'_{\mathtt{rel}} = V_{\mathtt{rel}}[X \mapsto V_{\mathtt{rel}}(X) \sqcup V_s \sqcup (o_{\mathtt{W}} \sqsupseteq \mathtt{rel}\,?\,V'_{\mathtt{cur}})] \\ V_{\mathtt{m}} = (o_{\mathtt{W}} \sqsupseteq \mathtt{rlx}\,?\,(V'_{\mathtt{rel}}(X) \sqcup V_r)) \end{array}}{\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle \xrightarrow{o_{\mathtt{W}}, V_r, m}_{\mathtt{W}} \langle V'_{\mathtt{rel}}, V'_{\mathtt{cur}}, V'_{\mathtt{acq}}\rangle}$$

(RACE: PROMISE)

$$\frac{X \in P_G \setminus P}{\mathsf{race}(\mathcal{V}, P, P_G, M, X, o)}$$

(RACE: MESSAGE)

$$\frac{\langle X@(\_,t], \_, o_{\mathtt{W}}, \_\rangle \in M \qquad \mathcal{V}.\mathtt{cur}(X) < t \qquad o_{\mathtt{W}} = \mathtt{na} \vee o = \mathtt{na}}{\mathsf{race}(\mathcal{V}, P, P_G, M, X, o)}$$

(FULFILL-HELPER)

$$\frac{X \in P}{\langle P, P_G\rangle \xrightarrow{X, o_{\mathtt{W}}} \langle P \setminus \{X\}, P_G \setminus \{X\}\rangle}$$

(FENCE-HELPER: NON-SC)

$$\frac{\mathcal{V}' = \begin{cases} \langle V_{\mathtt{rel}}, V_{\mathtt{acq}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{acq} \\ \langle \lambda\_.V_{\mathtt{cur}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{rel} \\ \langle \lambda\_.V_{\mathtt{acq}}, V_{\mathtt{acq}}, V_{\mathtt{acq}}\rangle & o_{\mathtt{F}} = \mathtt{acqrel} \end{cases}}{\langle\langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle, S\rangle \xrightarrow{o_{\mathtt{F}}}_{\mathtt{F}} \langle \mathcal{V}', S\rangle}$$

(FENCE-HELPER: SC)

$$\frac{S' = \mathcal{V}.\mathtt{acq} \sqcup S \qquad \mathcal{V}' = \langle \lambda\_.S', S', S'\rangle}{\langle \mathcal{V}, S\rangle \xrightarrow{\mathtt{sc}}_{\mathtt{F}} \langle \mathcal{V}', S'\rangle}$$

Figure IV.10: The full $\mathrm{PS}^{\mathtt{IR}}$ model (domains and auxiliary definitions).

Thread Configuration Steps Thread configuration steps are defined in Fig. IV.11. PROMISE, SILENT, READ, and WRITE steps are naturally extended from the simple $\mathrm{PS}^{\mathtt{IR}}$ model presented in Fig. IV.5. Moreover, UPDATE, FENCE, and SYSTEM CALL steps are defined in a similar way to those steps in the full vRC11 model. Accordingly, we explain the remaining rules for thread step.

RESERVE and **CANCEL**. A thread can *reserve* an interval $(f,t]$ of a location X by simply adding a reservation $\langle X@(f,t]\rangle$ to its reservation set and the memory. Moreover,

(PROMISE)

$$\frac{X \notin P_G \qquad P' = P \cup \{X\} \qquad {P_G}' = P_G \cup \{X\}}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma, \mathcal{V}, P', R\rangle, {P_G}', S, M\rangle}$$

(RESERVE)

$$\frac{r = \langle X@(f, t]\rangle \qquad f < t \qquad M \# r \qquad R' = R \cup \{r\} \qquad M' = M \cup \{r\}}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma, \mathcal{V}, P, R'\rangle, P_G, S, M'\rangle}$$

(CANCEL)

$$\frac{r \in R \qquad R' = R \setminus \{r\} \qquad M' = M \setminus \{r\}}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma, \mathcal{V}, P, R'\rangle, P_G, S, M'\rangle}$$

(SILENT)

$$\frac{\sigma \to \sigma'}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}, P, R\rangle, P_G, S, M\rangle}$$

(READ)

$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, v)} \sigma' \qquad m = \langle X@(\_, \_], v, \_, \_\rangle \in M \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m}_\mathsf{R} \mathcal{V}'}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P, R\rangle, P_G, S, M\rangle}$$

(WRITE)

$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, v)} \sigma' \qquad m = \langle X@(\_, \_], v, \_, \_\rangle \qquad M \# m \qquad \mathcal{V} \xrightarrow{o_\mathsf{W}, \lambda\_.0, m}_\mathsf{W} \mathcal{V}' \qquad M' = M \cup \{m\} \qquad \langle P, P_G\rangle \xrightarrow{X, o_\mathsf{W}}{}^{?} \langle P', {P_G}'\rangle}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P', R\rangle, {P_G}', S, M'\rangle}$$

(UPDATE)

$$\frac{\sigma \xrightarrow{\mathsf{RMW}(X, o_\mathsf{R}, o_\mathsf{W}, v_r, v_w)} \sigma' \qquad m_r = \langle X@(\_, t_r], v_r, \_, V_r\rangle \in M \qquad m_w = \langle X@(t_r, t_w], v_w, \_, \_\rangle \qquad M \# m_w \qquad \mathcal{V} \xrightarrow{o_\mathsf{R}, m_r}_\mathsf{R} \xrightarrow{o_\mathsf{W}, V_r, m_w}_\mathsf{W} \mathcal{V}' \qquad M' = M \cup \{m\} \qquad \langle P, P_G\rangle \xrightarrow{X, o_\mathsf{W}}{}^{?} \langle P', {P_G}'\rangle}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P', R\rangle, {P_G}', S, M'\rangle}$$

(RACY-READ)

$$\frac{\sigma \xrightarrow{\mathsf{R}(X, o_\mathsf{R}, \mathtt{undef})} \sigma' \qquad \mathsf{race}(\mathcal{V}, P, P_G, M, X, o_\mathsf{R})}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}, P, R\rangle, P_G, S, M\rangle}$$

(RACY-WRITE)

$$\frac{\sigma \xrightarrow{\mathsf{W}(X, o_\mathsf{W}, \_)} \_ \qquad \mathsf{race}(\mathcal{V}, P, P_G, M, X, o_\mathsf{W})}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \bot, \mathcal{V}, \emptyset, R\rangle, P_G, S, M\rangle}$$

(RACY-UPDATE)

$$\frac{\sigma \xrightarrow{\mathsf{RMW}(X, o_\mathsf{R}, o_\mathsf{W}, \_, \_)} \_ \qquad o_\mathsf{R} = \mathtt{na} \vee o_\mathsf{W} = \mathtt{na} \vee \mathsf{race}(\mathcal{V}, P, P_G, M, X, \mathtt{rlx})}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \bot, \mathcal{V}, \emptyset, R\rangle, P_G, S, M\rangle}$$

(FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F}(o_\mathsf{F})} \sigma' \qquad o_\mathsf{F} \sqsupseteq \mathtt{sc} \Rightarrow P = \emptyset \qquad \langle \mathcal{V}, S\rangle \xrightarrow{o_\mathsf{F}}_\mathsf{F} \langle \mathcal{V}', S'\rangle}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \to \langle\langle \sigma', \mathcal{V}', P, R\rangle, P_G, S', M\rangle}$$

(SYSTEM CALL)

$$\frac{\sigma \xrightarrow{\mathsf{Sys}(e)} \sigma' \qquad P = \emptyset \qquad \langle \mathcal{V}, S\rangle \xrightarrow{\mathtt{sc}}_\mathsf{F} \langle \mathcal{V}', S'\rangle}{\langle\langle \sigma, \mathcal{V}, P, R\rangle, P_G, S, M\rangle \xrightarrow{\mathsf{Sys}(e)} \langle\langle \sigma', \mathcal{V}', P, R\rangle, P_G, S', M\rangle}$$

Figure IV.11: The full $\mathrm{PS}^{\mathtt{IR}}$ model (thread transitions)

(MACHINE: NORMAL)

$$\frac{\langle \mathcal{T}(\tau), P_G, S, M\rangle \to^* \xrightarrow{l} \langle T', {P_G}', S', M'\rangle \quad \langle T', P_G, S, \hat{M}\rangle \to^* \langle\langle \_, \_, \emptyset, \_\rangle, \_, \_, \_\rangle}{\langle \mathcal{T}, P_G, S, M\rangle \xrightarrow{l} \langle \mathcal{T}[\tau \mapsto T'], {P_G}', S', M'\rangle}$$

(MACHINE: UB)

$$\frac{\langle \mathcal{T}(\tau), P_G, S, M\rangle \to^+ \langle\langle \bot, \_, \_\rangle, {P_G}', S', M'\rangle}{\langle \mathcal{T}, P_G, S, M\rangle \to \langle \bot, {P_G}', S', M'\rangle}$$

Figure IV.12: The full PS^{IR} model (machine transitions)

the thread can *cancel* its reservation by removing it from its reservation set and the memory, so the thread can add other messages to the reserved interval.

RACY-READ/WRITE/UPDATE. Here, RACY-WRITE and RACY-UPDATE steps may race with a promise unlike these steps in the simple model. Note that there is no difference whether we allow these steps to race with a promise since the thread that promised can always perform a non-atomic write to fulfill the promise.

Consistency At every machine step, the thread taking the step should certify its promises against a *capped memory* that abstracts the most restrictive possible future memory. A capped memory $\hat{M}$ of a memory M is given by:

1. For every $m_1, m_2 \in M$ where $m_1.\texttt{loc} = m_2.\texttt{loc} = X$, $m_1.\texttt{to} < m_2.\texttt{from}$, and there is no message $m' \in M(X)$ such that $m_1.\texttt{to} < m'.\texttt{to} < m_2.\texttt{to}$, we include a reservation $\langle X@(m_1.\texttt{to}, m_2.\texttt{from}]\rangle$ to $\hat{M}$.

2. To each location X, we include a reservation $\langle X@(t_{max}, t_{max} + 1]\rangle$ to $\hat{M}$ where t_{max} is the maximal to-timestamp among the messages to X.

Note that, given a memory M, there always exists a unique capped memory of M. A thread configuration $\langle T, P_G, S, M\rangle$ is *consistent* if there exist T', ${P_G}'$, S', and M' such that:

$$\langle T, P_G, S, \hat{M}\rangle \to^* \langle T', {P_G}', S', M'\rangle \ \wedge \ T'.\texttt{prm} = \emptyset$$

where $T.\texttt{prm}$ denotes the set of promises of a thread state T.

Machine Steps Figure IV.9 presents the machine steps, which are defined just like in the simple model presented in Fig. IV.4.

16 Mapping to Hardware

In this section, we consider the compiler mapping of PS^{IR} to hardware: we present the proposed addition of "strong stores" to hardware models (§16.1); discuss the implementation of strong stores in existing hardware (§16.2); establish soundness of mapping PS^{IR} to the extended models (§16.3); and discuss a load-store fence instruction, an alternative to strong stores (§16.4).

16.1 Strong Stores in Hardware Models

We propose a new store instruction called a "strong store" that preserves load-store ordering in modern architectures. Strong stores are stronger than a plain hardware stores but weaker than release stores (or than "lightweight fence", `lwsync`, followed by a plain store, as release stores are implemented on Power). Next, we describe the proposed extension of the Armv8 and Power models.

Armv8 We define Armv8S as the extension of the Armv8 memory model [58, 4] with strong stores. The Armv8 memory model defines a relation called *barrier-ordered-before* (`bob`), modeling thread-local order of memory accesses that is induced by barriers and release/acquire accesses. For example, `bob` includes $\texttt{po};[\texttt{L}]$ that corresponds to the fact that a release store (denoted by `L`) is never reordered with an earlier instruction in the program order (denoted by `po`). In Armv8S, we extend `bob` to include also $[\texttt{R}];\texttt{po};[\texttt{S}]$, where `S` represents the set of strong stores. This simple modification enforces the preservation of the order of any load followed by a strong store in the program order.

Power Similarly to Armv8S, we define PowerS by extending the Power memory model of [5]. Specifically, we propose a modest extension of the "no-thin-air" rule of the Power consistency predicate that requires acyclicity of $\texttt{ppo} \cup \texttt{fence} \cup \texttt{rfe}$ to include $[\texttt{R}];\texttt{po};[\texttt{S}]$ as well. Roughly, this constraint forbids load buffering behaviors when the order of the load followed by the store is preserved by certain dependencies (`ppo`) or fences (`fence`). The PowerS model extends this rule to prevent the load-store reordering also when the order is preserved by a strong store.

16.2 Implementing Strong Stores on Existing Hardware

As discussed in §12.2, the weak behavior of LB-NA has been rarely observed in practice, despite massive testing on CPU implementations of multiple Arm and Power architectures. In particular, among the Armv8 and Power implementations that have been tested in [5, 4], only Qualcomm's Snapdragon 820 processor exhibited the load buffering behavior.

To gain more knowledge about the Snapdragon anomaly (and extend the dataset of [4]), we experimented with a new Snapdragon version. We acquired a Snapdragon 888 (SM8350) processor, and using the Litmus7 (part of DIY7) testing framework, we ran the 23 basic behavior tests.[13] Like other processors and unlike Snapdragon 820, Snapdragon 888 did not exhibit the weak behavior of LB-NA in 6000M runs. For comparison, weak behaviors of the well-known store buffering (SB) and message passing (MP) tests were observed in 93% and 0.676% (respectively) of the 6000M runs. The supplementary material [41] includes the full results for Snapdragon 888.

On all those implementations that do not exhibit load buffering, we believe that strong stores could be implemented *just like plain stores*, without any additional overhead. To validate this claim, we used the Herd7 model checker. We started from the available tests in the suite of [5, 4], which includes 3,773 tests for Armv8[14] and 3,116 tests for Power,[15] and confirmed that all behaviors that are forbidden by the strengthened hardware models where every store is strong (*i.e.*, the models obtained by including `[R];po;[W]` in the `bob` relation of Armv8 or the "no-thin-air" constraint of Power), but allowed by the existing models, were never observed on an implementation (except for Snapdragon 820). The supplementary material [41] includes the `cat` files defining the models, the tests we ran, and the logs of the results.

[13] `http://gallium.inria.fr/~maranget/cats7/model-aarch64/tests.html` [Accessed November 2022].

[14] `https://gallium.inria.fr/~maranget/cats7/model-aarch64/index.html` [Accessed November 2022].

[15] `https://gallium.inria.fr/~maranget/cats7/ppc9/` [Accessed November 2022].

16.3 Mapping PS$^{\mathtt{IR}}$ to Hardware

Given strong stores in hardware, the mapping of PS$^{\mathtt{IR}}$ to hardware follows the standard schemes of C/C++ concurrency primitives,[16] except that relaxed writes in PS$^{\mathtt{IR}}$ are mapped to *strong* stores (while non-atomic writes are compiled to plain stores). We do not assume here that the hardware generally preserves load-store ordering, in which case, strong stores are not needed at all. In addition, the soundness of the "short-term" solution (see §12.2), which, in the absence of strong stores, suggests mapping relaxed writes as if they were release, follows from the discussion below since release writes are mapped to constructs that provide stronger guarantees than strong stores.

Remark 11. As was observed in [17], to be able to match every out-of-order execution to an in-order racy execution (which we need for Thm. 11, and Cho et al. [17] need for LDRF-PF), PS$^{\mathtt{IR}}$ has to forbid the reordering of RMWs with subsequent writes. Then, the mapping of certain RMW instructions to Armv8 requires an extra "fake" control dependency from the read part of the RMW, so the hardware will not reorder RMWs with following plain writes (which arise from non-atomic writes in the source). We refer the reader to [17] for the exact mapping scheme and the (unnoticeable) performance impact of it. We note that for hardware that preserves load-store ordering for all stores, this additional fake dependency is not needed.

To formally state the correctness of this mapping, since there are no system calls in the hardware models, we define the set of *outcomes* of a program for representing the final memories obtained after program executions are completed. This notion is defined for PS$^{\mathtt{IR}}$ and Armv8S as follows.

Definition 2. A function $o : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a program prog under* PS$^{\mathtt{IR}}$ if some execution of *prog* terminates with a memory M (*i.e.*, $\mathsf{init}(prog) \to^* \langle \mathcal{T}, P_G, M\rangle \wedge \mathsf{terminal}(\langle \mathcal{T}, P_G, M\rangle)$), and $o(X) = v$ where $\langle X@t, v, \_, \_\rangle \in M$ is the message to X with the greatest timestamp t.

Definition 3. A function $o : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a program prog under* Armv8S (PowerS) if o assigns to every location X the value of the co-maximal write to

[16] http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html [Accessed November 2022].

X in some execution graph of *prog* that is Armv8S-consistent (PowerS-consistent).[17]

Using these definitions, the soundness of mapping from PS^{IR} to Armv8S is stated as follows.

Theorem 12. For a PS^{IR} program *prog*, we denote by $(\!|prog|\!)_{\text{A}}$ the Armv8S program obtained by mapping *prog* as described above. Then, given a program *prog* and an outcome o of $(\!|prog|\!)_{\text{A}}$ under Armv8S, we have that either o is an outcome of *prog* under PS^{IR} or *prog* has undefined behavior under PS^{IR} (*i.e.*, it has an execution reaching a machine state of the form $\langle \bot, \_, \_ \rangle$).

To prove this theorem, we utilized the operational model for Armv8 by Pulte et al. [59], who also showed (in Coq) its equivalence to the declarative formulation of Armv8. We extended their operational model with strong relaxed accesses, reestablished the equivalence of the extended models, and proved (in Coq), using a simulation argument, that runs of this extended operational model reaching a certain outcome corresponds to runs of PS^{IR} that yield the same outcome. Note that our result natually applies to x86-TSO as the operational model for x86-TSO by Cho et al. [16] is stronger than Armv8S (operational) model where every store is considered as a strong store.

We believe the standard mapping from PS^{IR} to PowerS (with relaxed stores compiled as strong stores) is sound as well. Formally establishing the soundness of this mapping, possibly using the IMM memory model [57], is left as future work.

16.4 Load-store Fences Instead of Strong Stores

As an alternative to strong stores, we may introduce a load-store fence that preserves the order between all preceding loads with all succeeding stores. With this fence instruction, a relaxed write in PS^{IR} can be mapped to a load-store fence followed by a plain hardware store. While strong stores may give a more fine-grained control, load-store fences have their own benefits. First, they can reduce the pressure to the instruction set space. Typically, the bit representations of store instructions should encode lots of information such as a memory address, a value to be stored, an immediate, a memory ordering, and so on. Therefore, introducing a new store ordering

[17] Intuitively, the coherence order (co), which totally orders the writes to each location, corresponds to the timestamp order in PS^{IR}.

(*i.e.*, a strong store) may consume a significant portion of the instruction space. On the other hand, introducing a fence instruction will require only a small set of bit representations,[18] and thus, it will reduce the pressure to the instruction space. Second, once load-store fences are used for mapping relaxed writes, we do not need an extra control dependency for compiling a relaxed RMW instruction to Armv8 architecture. Indeed, the mapping of a relaxed RMW instruction will place a load-store fence before the store part of the RMW, which effectively prevents succeeding stores from being reordered with the load part of the RMW. Note that the mapping of a release RMW still requires an additional fake dependency (or placement of a loas-store fence before the store part of the RMW).

17 Proofs

In this section, we provide the pen-and-paper proofs of the results relating vRC11 to declarative models. §17.1 presents the proof of equivalence between the operational and declarative presentations of vRC11. Then, we prove Thm. 10 that states vRC11 is stronger than RC11 in §17.2. Note that all the proofs in this section are for the full vRC11 model including RMWs, fences, and release sequences.

17.1 Equivalence Between vRC11 and the Declarative Presentation

We prove the equivalence between vRC11 and its declarative presentation given in §13.2 relying on the existing proof of the equivalence between the promise-free fragment of the promising semantics and its declarative presentation by [33]. To distinguish the declarative presentation of vRC11 from vRC11 itself, we call the declarative model $\text{vRC11}_{\texttt{Axiom}}$.

Before proving the equivalence, we define the following auxiliary relation:

$$\texttt{rel} = ([\mathsf{W}_{\texttt{ra}}] \cup [\mathsf{F}_{\sqsupseteq\texttt{rel}}] ; \texttt{po}) ; \texttt{rs} \qquad (\textit{to-be-released})$$

Then, sw can be expressed as follows:

$$\texttt{sw} = \texttt{rel} ; \texttt{rf} ; ([\mathsf{R}_{\texttt{ra}}] \cup [\mathsf{R}_{\sqsupseteq\texttt{rlx}}] ; \texttt{po} ; [\mathsf{F}_{\sqsupseteq\texttt{acq}}]) \qquad (\textit{sync})$$

[18] In RISC-V, there already is an instruction reserved for the load-store fence [69]!

Next, we prove that vRC11 is stronger than $\text{vRC11}_{\texttt{Axiom}}$ using the *declarative (operational) machine* of $\text{vRC11}_{\texttt{Axiom}}$ as in [33]. We extend the declarative machine of [33] by adding the following race transition that yields UB:

$$\frac{G \text{ is vRC11}_{\texttt{Axiom}}\text{-racy}}{\langle \Sigma, G\rangle \to \langle \bot, G\rangle} \text{(RACE)}$$

Note that we assume that the sets of all behaviors of a program *prog* in $\text{vRC11}_{\texttt{Axiom}}$ and RC11, denoted by $[\![prog]\!]_{\text{vRC11}_{\texttt{Axiom}}}$ and $[\![prog]\!]_{\text{RC11}}$, are defined similarly to vRC11 using the declarative machine of $\text{vRC11}_{\texttt{Axiom}}$ and RC11, respectively.

We use the standard simulation technique to show the equivalence between vRC11 and $\text{vRC11}_{\texttt{Axiom}}$. In the following, we define the simulation relation between the two machines that slightly extends the relation provided in [33, Appendix B].

Definition 4. A *timestamp assignment* for an execution G is a function $f : \mathtt{W} \to \mathsf{Time}$. A timestamp assignment f is extended for sets of write events by $f(A) = \max_{a \in A} f(a)$.

Definition 5. An execution G induces the following additional derived relations: $G.\mathtt{rwr} = (\mathtt{rf}^? ; \mathtt{hb} ; [\mathtt{F}^{\mathtt{sc}}])^? ; (\mathtt{sc} ; [\mathtt{F}])^? ; \mathtt{hb}^? \cup (\mathtt{rf} ; \mathtt{hb}^?)$.

Definition 6. An $\text{vRC11}_{\texttt{Axiom}}$ machine state $\langle \Sigma, G\rangle$ *relates* to a vRC11 machine state $\mathcal{M} = \langle \mathcal{T}, \mathcal{S}, M\rangle$, denoted by $\langle \Sigma, G\rangle \sim \mathcal{M}$, if the following hold:

- G is coherent.
- $\mathcal{M}$ is well-formed.
- $\Sigma(i) = \mathcal{T}(i).\mathtt{st}$ for every $i \in \mathsf{Tid}$.
- There exists two timestamp assignments $f_{\mathtt{from}}$ and $f_{\mathtt{to}}$ for G for which the following properties hold:
  - For every $X \in \mathsf{Loc}$ and $a, b \in \mathtt{W}_X$, we have $f_{\mathtt{to}}(a) < f_{\mathtt{to}}(b)$ iff $\langle a, b\rangle \in \mathtt{mo}$.
  - For every $X \in \mathsf{Loc}$ and $a, b \in \mathtt{W}_X$, if $\langle a, b\rangle \in \mathtt{mo} \setminus \mathtt{rf} ; \mathtt{rmw}$, then $f_{\mathtt{to}}(a) \neq f_{\mathtt{from}}(b)$.
  - For every $b \in \mathtt{W}$, if $\langle a, b\rangle \notin \mathtt{mo} \setminus \mathtt{rf} ; \mathtt{rmw}$ for all a, then $f_{\mathtt{from}}(b) \neq 0$.
  - For every $X \in \mathsf{Loc}$, $M(X) = \{m_b \mid b \in \mathtt{W}_X\}$, where each m_b satisfies:

  - $m_b.\mathtt{val} = \mathtt{val}(b)$.
  - $m_b.\mathtt{to} = f_{\mathtt{to}}(b)$ and $m_b.\mathtt{from} = f_{\mathtt{from}}(b)$.
  - $m_b.\mathtt{from} = f_{\mathtt{to}}(a)$ if $\langle a, b\rangle \in \mathtt{rf}\,;\,\mathtt{rmw}$.
  - $m_b.\mathtt{mod} = \mathtt{mod}(b)$.
  - For every $y \in \mathsf{Loc}$, $m_b.\mathtt{view}(y) = f_{\mathtt{to}}(\{a \in \mathtt{W}_y \mid \langle a, b\rangle \in \mathtt{rwr}\,;\,\mathtt{rel}\})$.
  - If $b \in \mathit{range}(\mathtt{rmw})$, $\mathtt{mod}(b) \sqsupseteq \mathtt{rlx}$.
- For every $x \in \mathsf{Loc}$, $\mathcal{S}(X) = f_{\mathtt{to}}(\mathtt{W}_X^{\mathtt{sc}} \cup \mathit{dom}([\mathtt{W}_X]\,;\,\mathtt{rf}^?\,;\,\mathtt{hb}\,;\,[\mathtt{F}^{\mathtt{sc}}]))$.
- For every $i \in \mathsf{Tid}$, $\mathcal{T}(i) = \langle \Sigma(i), \mathcal{V}_i, \emptyset\rangle$ where $\mathcal{V}_i$ satisfies the following conditions for every $X, Y \in \mathsf{Loc}$:
  - $\mathcal{V}_i.\mathtt{rel}(Y)(X) = f_{\mathtt{to}}(\mathit{dom}([\mathtt{W}_X]\,;\,\mathtt{rwr}\,;\,[\mathtt{W}_Y^{\sqsupseteq \mathtt{ra}} \cup \mathtt{F}^{\mathtt{rel}}]\,;\,[\mathtt{E}_i]))$.
  - $\mathcal{V}_i.\mathtt{cur}(X) = f_{\mathtt{to}}(\mathit{dom}([\mathtt{W}_X]\,;\,\mathtt{rwr}\,;\,[\mathtt{E}_i]))$.
  - $\mathcal{V}_i.\mathtt{acq}(X) = f_{\mathtt{to}}(\mathit{dom}([\mathtt{W}_X]\,;\,\mathtt{rwr}\,;\,(\mathtt{rel}\,;\,\mathtt{rf}\,;\,[\mathtt{R}^{\sqsupseteq \mathtt{rlx}}])^?\,;\,[\mathtt{E}_i]))$.

Using the simulation relation given in Def. 6, we first prove that vRC11 is stronger than vRC11$_{\mathtt{Axiom}}$.

Lemma 4. Suppose that $\langle \Sigma, G\rangle \sim \mathcal{M}$. If $\mathcal{M}$ takes a non-racy transition $\mathcal{M} \to \mathcal{M}'$ in vRC11, there exists Σ' and G' such that

1. the vRC11$_{\mathtt{Axiom}}$ machine state takes the same transition $\langle \Sigma, G\rangle \to \langle \Sigma', G'\rangle$ in vRC11$_{\mathtt{Axiom}}$; and
2. $\langle \Sigma', G'\rangle \sim \mathcal{M}'$.

Proof. It directly follows from the simulation proof done by [33]. □

Lemma 5. Suppose that $\langle \Sigma, G\rangle \sim \langle \mathcal{T}, S, M\rangle$. If $\langle \mathcal{T}, S, M\rangle$ takes a racy transition $\langle \mathcal{T}, S, M\rangle \to \mathcal{M}'$ in vRC11, the vRC11$_{\mathtt{Axiom}}$ machine takes a racy transition in vRC11$_{\mathtt{Axiom}}$.

Proof. Suppose that a thread τ takes a transition racy at a location X:

$$\langle\langle \sigma, \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle\rangle, S, M\rangle \to \langle\langle \bot, \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle\rangle, S, M\rangle$$

where $\mathcal{T}(\tau) = \langle \sigma, \langle V_{\mathtt{rel}}, V_{\mathtt{cur}}, V_{\mathtt{acq}}\rangle\rangle$ and the program state σ transitions by accessing X with the access mode o. It means that there is a message $\langle X@(\_, t], \_, o_{\mathtt{W}}, \_\rangle \in M$ such

that $V_{\mathtt{cur}}(X) < t$ and either $o_{\mathtt{W}} = \mathtt{na}$ or $o = \mathtt{na}$. From the simulation relation on the memory, there is an event w in G such that $t = f_{\mathtt{to}}(w)$ and $\mathtt{mod}(w) = o_{\mathtt{W}}$. Moreover, from the simulation relation on the current view and the fact $V_{\mathtt{cur}}(X) < t$, we obtain that there is no event $e \in \mathsf{E}_\tau$ such that $\langle w, e\rangle \in G.\mathtt{rwr}$.

Now, we consider three cases where the access by σ is a read, write, or RMW. If the access is a read, an vRC11$_{\mathtt{Axiom}}$ machine state can always read from w, resulting in an vRC11$_{\mathtt{Axiom}}$-consistent graph G' by adding a new read event e_r with $\mathtt{loc}(e_r) = X$ and $\mathtt{mod}(e_r) = o$. Then, it is enough to show that $\langle w, e_r\rangle \in \mathtt{raceWW} \cup \mathtt{raceWR}$. First, since w is a write event, $\langle w, e_r\rangle \in [\mathtt{W}]\,;\mathtt{conflict}\,;[\mathtt{R}]$. Then, it is enough to show that $\langle w, e_r\rangle \notin \mathtt{pb} \cup \mathtt{exec}^{-1}$. Since the execution of vRC11$_{\mathtt{Axiom}}$ machine always add a $\mathtt{exec}$-maximal event, $\langle w, e_r\rangle \in \mathtt{exec}$, and thus, $\langle w, e_r\rangle \notin \mathtt{exec}^{-1}$. If $\langle w, e_r\rangle \in \mathtt{pb}$, there should be an event e' in G such that $\langle e', e_r\rangle \in \mathtt{po}$ and $\langle w, e'\rangle \in (\mathtt{rf}^?\,;\mathtt{hb}^?) \cup (\mathtt{rf}^?\,;\mathtt{hb}\,;\mathtt{sc}^?\,;\mathtt{hb}^?)$, which is not possible since it implies that $\langle w, e'\rangle \in G.\mathtt{rwr}$.

Next, if the access is a write, an vRC11$_{\mathtt{Axiom}}$ machine state can always add a $\mathtt{mo}$-maximal write. Then, the same argument as in the previous case applies here because $\mathtt{pb}$ does not depend on the $\mathtt{mo}$ relation.

Lastly, if the access is an RMW, it is necessary that $o_{\mathtt{W}} = \mathtt{na}$ (since there is no non-atomic RMW). From the simulation relation on the memory, the write w is not written by an RMW (*i.e.*, $b \notin \mathit{range}(\mathtt{rmw})$) since it has the access mode $\mathtt{mod}(w) = \mathtt{na}$. Therefore, the vRC11$_{\mathtt{Axiom}}$ machine state can read from the write that is the immediate predecessor of w in $\mathtt{mo}$, adding an event e_u, without breaking the atomicity of RMWs. Then, as in the read case, one can show that there is a race between w and e_u.

Therefore, if $\langle \mathcal{T}, S, M\rangle$ takes a racy transition in vRC11, so does $\langle \Sigma, G\rangle$ in vRC11$_{\mathtt{Axiom}}$. □

Lemma 6. For any program *prog*, $[\![\mathit{prog}]\!]_{\text{vRC11}} \subseteq [\![\mathit{prog}]\!]_{\text{vRC11}_{\mathtt{Axiom}}}$.

Proof. By induction on an execution of *prog* in vRC11. It follows from Lemma 4 and Lemma 5. □

Next, we prove the opposite direction, vRC11$_{\mathtt{Axiom}}$ is stronger than vRC11.

Lemma 7. Suppose that $\langle \Sigma, G\rangle \sim \mathcal{M}$. If $\langle \Sigma, G\rangle$ takes a non-racy transition $\langle \Sigma, G\rangle \to \langle \Sigma', G'\rangle$ in vRC11$_{\mathtt{Axiom}}$, there exists $\mathcal{M}'$ such that

1. the vRC11 machine state takes the same transition $\mathcal{M} \to \mathcal{M}'$ in vRC11; and
2. $\langle \Sigma', G' \rangle \sim \mathcal{M}'$.

Proof. As in Lemma 4, it directly follows from the simulation proof done by [33]. □

Lemma 8. Given a racy execution of a vRC11$_{\texttt{Axiom}}$ declarative machine starting from an initial machine state, $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma, G \rangle \to \langle \bot, G \rangle$, there exists a minimal racy execution towards machine states $\langle \Sigma_{min}, G_{min} \rangle$ and $\langle \Sigma_{race}, G_{race} \rangle$ such that

1. $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma_{min}, G_{min} \rangle \to \langle \Sigma_{race}, G_{race} \rangle$;
2. G_{race} is a sub-graph of G;
3. G_{min} is not vRC11$_{\texttt{Axiom}}$-racy;
4. G_{race} adds to G_{min} an event b that races with some event $a \in G_{min}$ by $\langle a, b \rangle \in \texttt{raceWR} \cup \texttt{raceWW}$; and
5. The sequence of system calls exhibited by these transitions is a prefix of the sequence exhibited by $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma, G \rangle$.

Proof. First, there exists the minimal racy prefix of the given execution $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma', G' \rangle \to \langle \Sigma'', G'' \rangle \to^* \langle \Sigma, G \rangle$ where G' is not vRC11$_{\texttt{Axiom}}$-racy and G'' is vRC11$_{\texttt{Axiom}}$-racy. Suppose that two events $a, b \in G''$ are racy (*i.e.*, $\langle a, b \rangle \in \texttt{raceWR} \cup \texttt{raceWW}$). Then, we can construct a sequence of transitions ending with a sub-graph G_{race} of G'' as follows:

(*i*) Starting from the initial machine $\langle \Sigma_0, G_0 \rangle$, take transitions to $\langle \Sigma_{<a}, G_{<a} \rangle$ that add every event related to a by $G''.\texttt{exec}$ $e \in G''.\texttt{exec}^{-1}(a) \triangleq \{ e \mid \langle e, a \rangle \in G''.\texttt{exec} \}$. (Note that $b \notin G''.\texttt{exec}^{-1}(a)$ since $\langle b, a \rangle \notin G''.\texttt{exec}^{-1}$.) Since given a vRC11$_{\texttt{Axiom}}$-consistent graph G, its every prefix in $\texttt{exec}$ order is also an vRC11$_{\texttt{Axiom}}$-consistent graph, this can be done by picking a $\texttt{exec}$-minimal event from $G''.\texttt{exec}^{-1}(a)$ and take a transition that adds the event.

(*ii*) Add a transition that adds a to the graph, resulting in an execution $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma_{<a}, G_{<a} \rangle \to \langle \Sigma_{\leq a}, G_{\leq a} \rangle$.

(*iii*) Similarly to (*i*), take transitions adding every event related to b by $\texttt{exec}$, resulting in an execution $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma_{<a}, G_{<a} \rangle \to \langle \Sigma_{\leq a}, G_{\leq a} \rangle \to^* \langle \Sigma_{min}, G_{min} \rangle$.

(*iv*) Finally, take another transition adding b, $\langle \Sigma_{min}, G_{min} \rangle \to \langle \Sigma_{race}, G_{race} \rangle$, we have the minimal racy execution satisfying *(1)-(4)*.

In particular, G_{race} is racy because $G_{race}.\texttt{pb} \cup G_{race}.\texttt{exec}^{-1} \subseteq G''.\texttt{pb} \cup G''.\texttt{exec}^{-1}$. Moreover, *(5)* is satisfied since the system call events are totally ordered by $G.\texttt{exec}$. □

Lemma 9. Given a minimal racy execution $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma_{min}, G_{min} \rangle \to \langle \Sigma_{race}, G_{race} \rangle$ satisfying *(1)-(4)* of Lemma 8 and an initial vRC11 machine state $\mathcal{M}_0$ related by $\langle \Sigma_0, G_0 \rangle \sim \mathcal{M}_0$, there exists $\mathcal{M}_{min}$ taking the same transitions and invoking UB, $\mathcal{M}_0 \to^* \mathcal{M}_{min} \to \langle \bot, S, M \rangle$.

Proof. From Lemma 7, there exists $\mathcal{M}_0 \to^* \mathcal{M}_{min}$ taking the same transitions as $\langle \Sigma_0, G_0 \rangle \to^* \langle \Sigma_{min}, G_{min} \rangle$ and $\langle \Sigma_{min}, G_{min} \rangle \sim \mathcal{M}_{min}$. It is enough to show that $\mathcal{M}_{min} = \langle \mathcal{T}_{min}, S_{min}, M_{min} \rangle$ can take a racy step to $\langle \bot, S, M \rangle$ for some S and M. Suppose that the last step in $\text{vRC11}_{\texttt{Axiom}}$, $\langle \Sigma_{min}, G_{min} \rangle \to \langle \Sigma_{race}, G_{race} \rangle$ adds an event b that races with some event $a \in G_{min}$ by $\langle a, b \rangle \in \texttt{raceWR} \cup \texttt{raceWW}$, where b is executed by a thread τ. From $\langle \Sigma_{min}, G_{min} \rangle \sim \mathcal{M}_{min}$, there exists a message $m_a \in M_{min}$ corresponding to a. Since a has never been propagated before b, $V_{\texttt{cur}}(\texttt{loc}(a)) < t_a = f_{\texttt{to}}(a)$ where $\mathcal{T}(\tau) = \langle \sigma, \langle V_{\texttt{rel}}, V_{\texttt{cur}}, V_{\texttt{acq}} \rangle \rangle$. Now, the program state σ of the thread τ is about to perform a memory access that corresponds to b. From the fact that $\langle a, b \rangle \in \texttt{conflict}$, the next transition of σ is accessing $\texttt{loc}(a)$ with an access mode $\texttt{mod}(b)$ where either one of $\texttt{mod}(a) = m_a.\texttt{mod}$ or $\texttt{mod}(b)$ is non-atomic. Then, the thread configuration $\langle \langle \sigma, \langle V_{\texttt{rel}}, V_{\texttt{cur}}, V_{\texttt{acq}} \rangle \rangle, S_{min}, M_{min} \rangle$ can take a racy transition that races with the message $m_a \in M_{min}$ and invoke UB. Therefore, $\mathcal{M}_{min} \to \langle \bot, S_{min}, M_{min} \rangle$. □

Lemma 10. For any program $prog$, $[\![prog]\!]_{\text{vRC11}_{\texttt{Axiom}}} \subseteq [\![prog]\!]_{\text{vRC11}}$.

Proof. It follows from Lemma 7, Lemma 8, and Lemma 9. □

Then, the equivalence between vRC11 and $\text{vRC11}_{\texttt{Axiom}}$ follows from Lemma 6 and Lemma 10.

Theorem 13. For any program $prog$, $[\![prog]\!]_{\text{vRC11}_{\texttt{Axiom}}} = [\![prog]\!]_{\text{vRC11}}$.

17.2 Relating vRC11 to RC11

We prove that vRC11 is stronger than RC11 by showing that $\text{vRC11}_{\texttt{Axiom}}$ is stronger than RC11.

The following derived relation is used in defining RC11:

$$\texttt{psc}_{\texttt{F}} = [\texttt{F}_{\texttt{sc}}] ; (\texttt{hb} \cup \texttt{hb} ; \texttt{eco} ; \texttt{hb}) ; [\texttt{F}_{\texttt{sc}}] \qquad \textit{(partial-SC-fence-order)}$$

Given the derived relations in §13.2 and above, a consistent execution in RC11 is defined as follows.

Definition 7. An execution graph G is RC11-*consistent* if the following hold:

- $\texttt{hb} ; \texttt{eco}^?$ is irreflexive. (RC11-COHCRENCE)
- $\texttt{psc}_{\texttt{F}}$ is acyclic. (RC11-SC-FENCE)
- $\texttt{rmw} \cap (\texttt{rb} ; \texttt{mo}) = \emptyset$. (RC11-ATOMICITY)
- $\texttt{po} \cup \texttt{rf}$ is acyclic. (RC11-NO-LB)

Then, a racy graph in RC11 is defined as follows.

Definition 8. An execution graph G is RC11-*racy* if $\texttt{conflict} \setminus (\texttt{hb} \cup \texttt{hb}^{-1}) \neq \emptyset$. A program *prog* has undefined behavior under RC11 if it has some racy RC11-consistent execution.

Next, we prove that $\text{vRC11}_{\texttt{Axiom}}$ is stronger than RC11 by showing that (*i*) an $\text{vRC11}_{\texttt{Axiom}}$-consistent graph is RC11-consistent; and (*ii*) an $\text{vRC11}_{\texttt{Axiom}}$-racy graph is RC11-racy.

Lemma 11. An $\text{vRC11}_{\texttt{Axiom}}$-consistent execution graph G is RC11-consistent.

Proof. We show that G satisfies the four axioms of RC11-consistency.

(RC11-COHERENCE) It follows from (COHERENCE) of $\text{vRC11}_{\texttt{Axiom}}$-consistency.

(RC11-SC-FENCE) Since $\mathtt{sc}$ is a strict total order on SC fences, it is enough to show that $\mathtt{psc_F} \subseteq \mathtt{sc}$. From (RC11-COHERENCE) that we have already proved, it is clear that $\mathtt{psc_F}$ is irreflexive. Then, suppose that there are two distinct SC fence events f_1 and f_2 such that $\langle f_1, f_2\rangle \in \mathtt{psc_F}$ and $\langle f_1, f_2\rangle \notin \mathtt{sc}$. Since $\mathtt{sc}$ is a total order, $\langle f_2, f_1\rangle \in \mathtt{sc}$. From (NO-LB) of vRC11$_{\mathtt{Axiom}}$, $\langle f_1, f_2\rangle \notin \mathtt{hb}$ since otherwise, $\mathtt{sc} \cup \mathtt{hb} \subseteq \mathtt{sc} \cup \mathtt{rf}$ becomes cyclic. Therefore, by replacing $\mathtt{eco}$ with an equivalent relation $\mathtt{rf} \cup (\mathtt{mo} \cup \mathtt{rb}); \mathtt{rf}^?$ [37] in $\mathtt{psc_F}$, we have $\langle f_1, f_2\rangle \in \mathtt{rf} \vee \langle f_1, f_2\rangle \in \mathtt{mo}\,;\,\mathtt{rf}^? \vee \langle f_1, f_2\rangle \in \mathtt{rb}\,;\,\mathtt{rf}^?$. If $\langle f_1, f_2\rangle \in \mathtt{rf}$, (NO-LB) of vRC11$_{\mathtt{Axiom}}$ is violated since $\langle f_1, f_1\rangle \in \mathtt{rf}\,;\,\mathtt{sc}$. For the other two cases, (SC-FENCE) of vRC11$_{\mathtt{Axiom}}$ is violated. Therefore, there should be no such SC fence events f_1 and f_2, and thus, $\mathtt{psc_F} \subseteq \mathtt{sc}$.

(RC11-ATOMICITY) It follows from (ATOMICITY) of vRC11$_{\mathtt{Axiom}}$-consistency.

(RC11-NO-LB) It follows from (NO-LB) of vRC11$_{\mathtt{Axiom}}$-consistency. □

Lemma 12. If an vRC11$_{\mathtt{Axiom}}$-consistent execution graph G is vRC11$_{\mathtt{Axiom}}$-racy, then it is also RC11-racy.

Proof. It suffices to show that $\mathtt{raceWW} \cup \mathtt{raceWR} \subseteq \mathtt{conflict} \setminus (\mathtt{hb} \cup \mathtt{hb}^{-1})$. From the definitions of the derived relations, we have $\mathtt{hb} \subseteq \mathtt{pb}$ and $\mathtt{hb} \subseteq (\mathtt{po} \cup \mathtt{rf})^+ \subseteq \mathtt{exec}$. Therefore, $\mathtt{raceWW} \cup \mathtt{raceWR} \subseteq \mathtt{conflict} \setminus (\mathtt{pb} \cup \mathtt{exec}^{-1}) \subseteq \mathtt{conflict} \setminus (\mathtt{hb} \cup \mathtt{hb}^{-1})$. □

The proof of Thm. 10.

Proof. Thm. 10 follows from Lemma 11, Lemma 12, and Thm. 13. □

18 Related Work

Our proposal for a concurrency semantics refines, simplifies, and combines existing ideas: catch-fire and preserving load-store ordering as in RC11 [11, 37, 12], an operational presentation of RC11 using the promising semantics without promises and certified promises as a speculation mechanism to allow load-store reordering [33], justifying compiler optimizations on non-atomics based on sequential reasoning [18]

(see also [70]), and the operational Arm model as a bridge between the high-level semantics and the hardware model [59]. We also rely on [5, 4] for the models of Power and Arm, the experimental data on observed behaviors, the Herd model checker, and the testing framework; and on [53] for the performance evaluation of different compilation schemes.

In particular, our $\mathrm{PS}^{\mathtt{IR}}$ model is inspired by the $\mathrm{PS}^{\mathtt{na}}$ model in [18]. The most significant difference between $\mathrm{PS}^{\mathtt{IR}}$ and $\mathrm{PS}^{\mathtt{na}}$ is that $\mathrm{PS}^{\mathtt{na}}$ allows also promises of *relaxed* writes, which makes $\mathrm{PS}^{\mathtt{na}}$ significantly more complex than $\mathrm{PS}^{\mathtt{IR}}$. First, in $\mathrm{PS}^{\mathtt{na}}$ a thread promises concrete messages with specific timestamp, value, and view, while $\mathrm{PS}^{\mathtt{IR}}$ only maintains sets of locations that threads will write to in the future. (This is possible because promises of $\mathrm{PS}^{\mathtt{IR}}$ are needed only for race detection.) Second, unlike $\mathrm{PS}^{\mathtt{IR}}$, $\mathrm{PS}^{\mathtt{na}}$ allows a thread to *modify* their promises by lowering or splitting them, complicating the model and the proofs substantially. Lastly, a non-atomic write in $\mathrm{PS}^{\mathtt{IR}}$ adds a single message to the memory, while in $\mathrm{PS}^{\mathtt{na}}$ multiple messages may be added by a single non-atomic write.

The idea to use an undefined value rather than catch-fire in order to validate load introduction comes from the LLVM (informal) model. To the best of our knowledge, the first attempt to apply this approach in a formal model was in [14], where previous read values can be revisited whenever relevant writes are executed. This requires a rather complicated event-structure-based model, which does not admit the DRF guarantee. Later improvements of this model [15, 52] admit DRF but *fail* to support load introduction. In turn, our $\mathrm{PS}^{\mathtt{IR}}$ model (following $\mathrm{PS}^{\mathtt{na}}$) applies this approach together with promises. We are not aware of any previous proof relating an in-order source model based on catch-fire to an IR model that is based on undefined values.

Other weak memory models were recently proposed (see, *e.g.*, [55, 26, 29]), but they are all focused on generally allowing load-store reordering, while our models (both source and IR) allow it only for non-atomic accesses. Notably, supporting load introduction in these models is rather hard, and besides the promising models (*e.g.*, the recent version in [18]), we are not aware of any model that fully supports load-store reordering as well as load introduction. In particular, Jeffrey et al. [29] observe a tension between the kind of temporal reasoning supported by their model and load introduction.

In contrast, other works, *e.g.*, [46, 50], propose SC as a concurrency semantics for programmers, and study its expected cost (which can be rather high). In turn, we believe that an in-order model enjoys the advantages of SC (*i.e.*, in-order reasoning), while allowing for rather minimal performance overhead, provided that catch-fire for races on non-atomics is acceptable.

Chapter V

Conclusion

This dissertation thoroughly investigated the conflicting desiderata for relaxed memory models, efficiency and usability, and proposed memory models that balance between them at the minimal price. First, we proposed PS2, the first relaxed memory model that provably validates inter-thread optimizations while admitting data-race-freedom guarantees. PS2 redesigned the promising semantics with two key ideas: capped memory and reservations. In particular, the reservation mechanism also solves the problem of inefficiency in mapping RMW instructions of the promising semantics to Armv8 architecture. Second, we developed vRC11, an in-order semantics for relaxed memory concurrency with only a negligible performance overhead. To validate all common compiler optimizations performed on non-atomic code, we utilized an out-of-order model PS^{IR}. Since PS^{IR} is based on PS2, PS^{IR} validates inter-thread optimizations allowed for the PS2 model. For atomic accesses, we proved that it is inevitable to prevent the reordering of a (non-atomic) read followed by a relaxed write. We observed that load-store reordering is only performed by few Arm CPU implementations and that the performance benefits of this reordering is negligible in the Arm architecture design. Accordingly, we proposed a new store instruction called a strong store in Arm for compiling relaxed writes with negligible overhead.

As this dissertation addressed the problem of defining a proper semantics for

relaxed memory concurrency, the next step forward is to facilitate formal methods in real-world concurrent programming. In the following, we discuss some of the future works in this direction.

Modeling unsupported memory orderings. While the models presented in this disseration include most concurrency features of C/C++, they are still missing *consume reads* and *sequentially consistent accesses*. Initially, the purpose of consume is to provide language support for read-copy-update (RCU) used in Linux kernel [51]. The idea of consume reads is to give a similar guarantee as acquire reads without any hardware overhead by assuring the ordering only when there is a preserved dependency from a consume read to a succeeding instruction. We may adapt the idea of register views in the promising semantics for Armv8 architecture [59] to model the memory order consume. While Lahav et al. [37] reported a flaw in C11 model for sequentially consistent (SC) accesses and proposed a remedy, the purpose of SC accesses and the desired guarantees for them are still remained vague. Therefore, the practical usage of SC accesses should be investigated before extending the models to include those accesses.

Extending memory models with more features. Most research in relaxed memory concurrency assumes only minimal concurrency features such as memory accesses (with different access modes) and fences while ignoring real-world programming language characteristics. In order to facilitate formal methods in real-world concurrent programming, common language features should be exhaustively captured by the memory model. Luckily, most of such features are orthogonal to the underlying concurrency semantics, so existing formal semantics for sequential programs such as CompCertC [45] can be easily adapted in relaxed memory models. However, some key programming features like dynamic memory allocation or mixed-size accesses require considerable support from the memory model to be properly modeled.

Dynamic memory allocation is a key feature in low-level programming languages like C/C++. However, modeling dynamic allocation in relaxed memory is challenging due to various compiler optimizations on `malloc` and `free`. For instance, to support reordering of a `free` instruction with earlier memory reads, the IR memory

model should allow a thread to "promise" a `free` instruction. To model dynamic allocation, one should investigate the programmers' assumption in concurrent programming, common compiler optimizations performed on `malloc` and `free`, and actual implementations of them.

Common architectures allow mixed-size accesses, *i.e.*, memory accesses with various sizes (*e.g.*, 4, 8, or 16 bytes). Notably, there has been no weak memory model for source-level languages that validates practical programming patterns in the Linux kernel that relies on mixed-size accesses and is efficiently mapped to hardware architectures like Armv8 [4]. Development of a source-level semantics for mixed-size concurrency remains a future work.

Developing a realistic verified compiler. Despite many years of research, no realistic verified compiler properly supports relaxed memory concurrency. The challenge is that verification of real-world compiler optimizations in relaxed memory models is extremely difficult. Indeed, previous approaches that build on CompCert, a verified C compiler for single-threaded programs, often gave up the optimization passes of CompCert even though they assume relatively simpler memory models such as TSO or sequential consistency [30, 68]. We believe, by extending the approach of [18], it should be possible to reuse most of the correctness proofs of CompCert passes without exposing the full complexity of the underlying relaxed memory model. There are a couple of challenges that should be addressed to extend the soundness of CompCert to relaxed memory. For example, the sequential machine by Cho et al. [18] should be generalized to validate memory optimizations that alter the memory layout (*e.g.*, merging memory allocations, register promition, and register spilling).

References

[1] 2023. Coq development and supplementary material for this dissertation. https://sf.snu.ac.kr/sunghwan.lee/thesis/

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, S. Krishna, and Viktor Vafeiadis. 2021. The Decidability of Verification under PS 2.0. In *ESOP*. Springer International Publishing, Cham, 1–29. https://doi.org/10.1007/978-3-030-72019-3_1

[3] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering–a New Definition. In *ISCA*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[4] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (July 2021), 54 pages. https://doi.org/10.1145/3458926

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[6] Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *POPL*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2429069.2429099

[7] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP*. Springer, Berlin, Heidelberg, 283–307. http://dx.doi.org/10.1007/978-3-662-46669-8_12

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[9] John Bender and Jens Palsberg. 2019. A Formalization of Java's Concurrent Access Modes. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 142:1–142:28. https://doi.org/10.1145/3360568

[10] Hans-Juergen Boehm. 2019. P1217R2: Out-of-thin-air, revisited, again. https://wg21.link/p1217 [Online; accessed 22-March-2020].

[11] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *PLDI*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[12] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding out-of-Thin-Air Results. In *MSPC*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

[13] C/C++11 Mappings to Processors 2021. Retrieved March 18, 2021 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[14] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO*. IEEE Press, 100–110. https://doi.org/10.1109/CGO.2017.7863732

[15] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 70 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290383

[16] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-X86 and Armv8. In *PLDI*. ACM, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

[17] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. In *PLDI*. ACM, New York, NY, USA, 867–882. https://doi.org/10.1145/3453483.3454082

[18] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency. In *PLDI*. ACM, New York, NY, USA, 213–228. https://doi.org/10.1145/3519939.3523718

[19] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Jan. 2020), 29 pages. https://doi.org/10.1145/3371102

[20] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *PLDI*. ACM, New York, NY, USA, 792–808. https://doi.org/10.1145/3519939.3523451

[21] Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *ESOP*. Springer International Publishing, Cham, 1027–1055. https://doi.org/10.1007/978-3-319-89884-1_36

[22] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 Programs Operationally. In *PPoPP*. ACM, New York, 355–365. https://doi.org/10.1145/3293883.3295702

[23] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*. Springer Berlin Heidelberg, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

[24] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

[25] INRIA. [n. d.]. The Coq Proof Assistant. http://coq.inria.fr/.

[26] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with Preconditions: A Simple Model of Relaxed Memory. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 194 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428262

[27] Alan Jeffrey and James Riely. 2016. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *LICS 2016*. ACM, 759–767.

[28] Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:33)2019

[29] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 6, POPL, Article 54 (Jan. 2022), 30 pages. https://doi.org/10.1145/3498716

[30] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *PLDI*. ACM, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

[31] JMM causality test cases 2019. Retrieved November 17, 2019 from http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html

[32] Jeehoon Kang. 2019. *Reconciling low-level features of C with compiler optimizations.* Ph. D. Dissertation. Seoul National University.

[33] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[34] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

[35] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *PLDI*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

[36] Ori Lahav and Roy Margalit. 2019. Robustness Against Release/Acquire Semantics. In *PLDI*. ACM, New York, NY, USA, 126–141. https://doi.org/10.1145/3314221.3314604

[37] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[38] Doug Lea. 2019. JEP 188: Java Memory Model Update. Retrieved November 17, 2019 from http://openjdk.java.net/jeps/188

[39] Doug Lea. 2019. Using JDK 9 Memory Order Modes. Retrieved November 17, 2019 from http://gee.cs.oswego.edu/dl/html/j9mm.html

[40] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. ACM, New York, NY, USA, 633–647. https://doi.org/10.1145/3062341.3062343

[41] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Coq development and supplementary material for the paper "Putting Weak Memory in Order via a Promising Intermediate Representation". https://sf.snu.ac.kr/promising-ir/

[42] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representations. *Proc. ACM Program. Lang.* 7, PLDI, Article 183 (June 2023), 24 pages. https://doi.org/10.1145/3591297

[43] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Coq development and supplementary material for the paper "Promising 2.0: Global Optimizations in Relaxed Memory Concurrency". http://sf.snu.ac.kr/promising2.0

[44] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI*. ACM, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

[45] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[46] Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2021. Safe-by-Default Concurrency for Modern Programming Languages. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 10 (Sept. 2021), 50 pages. https://doi.org/10.1145/3462206

[47] Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *ASPLOS*. ACM, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

[48] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[49] Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434285

[50] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *PLDI*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1993498.1993522

[51] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. 2017. P0190R4: Proposal for New `memory_order_consume` Definition. https://wg21.link/p0190 [Online; accessed 3-July-2023].

[52] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2020. Reconciling Event Structures with Modern Multiprocessors. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.5

[53] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

[54] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. Springer, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[55] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP*. Springer, Cham, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

[56] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *POPL*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

[57] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

[58] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

[59] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *PLDI*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

[60] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL, Article 68 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290381

[61] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *PLDI*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

[62] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *ISCA*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/3079856.3080206

[63] Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *Proc. ACM Program. Lang.* 7, POPL, Article 53 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571246

[64] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP*. Springer International Publishing, Cham, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[65] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *POPL*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

[66] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, New York, NY, USA, 867–884. https://doi.org/10.1145/2509136.2509532

[67] Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. Springer-Verlag, Berlin, Heidelberg, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

[68] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

[69] Andrew Waterman and Krste Asanović. 2017. The RISC-V Instruction Set Manual Volume I: User-Level ISA. https://riscv.org/specifications/isa-spec-pdf/

[70] Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *PLDI*. ACM, New York, NY, USA, 903–917. https://doi.org/10.1145/3519939.3523734

초록

느슨한 메모리 모델을 정의하는 것은 수십 년 간 프로그래밍 언어 분야의 중요한 난제로 여겨져 왔다. 문제는 느슨한 메모리 모델에 대한 두 가지 주요 요구사항인 사용성과 효율성이 첨예하게 대립하는 데에 기인한다. 사용성은 모델이 상식적이고 이해할 수 있어야 한다는 것으로, 프로그래머가 메모리 모델을 이용하여 동시성 프로그램을 개발하고 논증할 수 있도록 한다. 효율성은 모델을 효율적으로 컴파일 할 수 있어야 한다는 것으로, 메모리 모델이 일반적인 컴파일러 최적화를 허용하고, 하드웨어로도 효율적으로 컴파일될 수 있어야 함을 의미한다. 이 두 가지 요구사항은 느슨한 메모리 모델을 설계하는 데 있어 핵심적인 원칙이지만, 두 원칙을 모두 충족하는 메모리 모델을 정의하는 것은 매우 어려운 일이다.

본 학위논문에서는 느슨한 메모리 모델에 요구되는 성질을 깊게 이해하고, 여러 요구사항 사이의 본질적인 충돌을 발견하며, 이러한 충돌을 최소한의 비용으로 해결하는 느슨한 메모리 모델을 설계한다. 먼저 본 논문에서는 글로벌 분석에 기반한 컴파일러 최적화를 지원하는 최초의 느슨한 메모리 모델인 PS 2.0 모델을 제안한다. PS 2.0은 기존 모델인 Promising semantics (PS)의 핵심 요소를 새롭게 설계하여 데이터 경쟁 정리를 비롯한 PS에 대한 기존의 결과를 보장하면서도 글로벌 값 분석을 이용한 최적화 및 레지스터 프로모션을 지원한다. 또한, PS 2.0은 기존 PS의 RMW 연산을 Armv8 아키텍처로 컴파일 할 때 발생하는 비효율성 문제를 해결하였다. 두 번째로 프로그램 명령어를 순서대로 실행하면서도 사실상의 성능 저하를 수반하지 않는 쉽고 간단한 메모리 모델을 제안한다. 먼저, 순서대로 실행하는 프로그래밍 언어 모델과 순서를 바꾸어 실행하는 컴파일러 중간 언어 모델을 분리함으로써 프로그래머에게 간단한 모델을 제공하면서도 기존 컴파일러가 수행하는 모든 컴파일러 최적화를 지원하는 방법을 고안하였다. 하드웨어로 컴파일할 때는 일부 쓰기 명령어가 앞선 읽기 명령어와 순서가 바뀌어 실행되는 것을 방지해야 하는데, 이를 사실성의 성능 저하 없이 구현하는 방법을 소개한다.

주요어: 동시성, 느슨한 메모리 모델, 실행 모델, 컴파일러 최적화, 정형 기법
학번: 2017-23151

Acknowledgments

I would never be able to complete this dissertation without the help and support of many people. First and foremost, my advisor Prof. Chung-Kil Hur taught me how I should think as a researcher, and his enthusiasm always motivated and encouraged me. I always enjoyed working and discussing with Prof. Ori Lahav, from whom I learned how to present my ideas both in speech and writing. I am grateful for having your guidance throughout my Ph.D.

I would like to thank Prof. Jeehoon Kang, who gave me valuable advice in my research and career. I also thank Prof. Kwangkeun Yi and Prof. Jieung Kim for their feedback on this dissertation as thesis committee members. I always enjoyed spending time with SF and ROPAS lab members. In particular, I thank Minki Cho not only for working with me on many of my papers but also for inspiring me with his brilliant ideas. I also thank all my collaborators who worked with me on papers on which this dissertation is based.

I thank my parents for their endless support. Without what they have provided since my birth, I would never be what I am today. I have also watched my sister's effort toward her career path. I always believe that she will achieve her goals shortly. Finally, I dedicate this dissertation to Hyejoo, who has stood by my side for almost entire years of my Ph.D. journey without losing her faith in me.