Ph.D. DISSERTATION

Simplifying Reasoning under Weak Memory Concurrency

느슨한 메모리 프로그램을 쉽게 이해하기

August 2023

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

Minki Cho

Simplifying Reasoning under Weak Memory Concurrency

느슨한 메모리 프로그램을 쉽게 이해하기

지도교수 허충길

이 논문을 공학박사 학위논문으로 제출함

2023년 7월

서울대학교 대학원

컴퓨터공학부

조 민 기

조민기의 공학박사 학위논문을 인준함

2023년 7월

위 원 장	이 광 근	[illegible] (인)
부위원장	허 충 길	[illegible] (인)
위 원	Ori Lahav	[illegible] (인)
위 원	강 지 훈	[illegible] (인)
위 원	김 지 웅	[illegible] (인)

Abstract

This thesis presents ways to easily implement concurrent programs in weak memory. In shared memory concurrency, programs have counterintuitive (or *weak*) behavior due to hardware and compiler optimizations. It is difficult to implement a correct program with a full understanding of weak behavior. I develop two theories that simplify reasoning under weak memory concurrency. First, I formalize and prove local data-race-freedom guarantees that ensure strong semantics for locations accessed by non-racy instructions. These allow programmers who avoid data races to understand and write concurrent programs without understanding of weak behavior. Next, I show that sequential reasoning is adequate and sufficient for establishing soundness of compiler optimizations under weak memory. I introduce a sequential model SEQ which has no weak behavior and no concurrency, and show that correct optimizations under SEQ executions are also correct under weak memory. Our results are fully verified against a weak memory model, the promising semantics.

Keywords: Relaxed Memory Concurrency, Operational Semantics, Data Race Freedom, Compiler Optimization, Verified Compilation

Student Number: 2017-29003

Contents

Chapter 1

Prologue

1.1 Introduction

Writing and compiling programs correctly under shared memory concurrency is difficult. In shared memory concurrency, programs have counterintuitive (or “weak”) behavior due to hardware and compiler optimizations. Weak memory models are introduced to abstract complicated weak behavior and allow programmers to write correct programs without sacrificing optimizations.

However, despite recent advances, weak memory models are too complex. Weak memory models introduce complicated states and out-of-order execution to properly model weak behavior. These complexities dramatically increase the number of cases to consider when reasoning about program execution.

In this thesis, I propose methods to simplify reasoning under weak memory concurrency. The main idea is to provide a simple and understandable model for programmers instead of the full weak memory model. Importantly, reasoning from the simple model can be *lifted* to the full weak memory model. Thanks to this property, programmers can reason correctly about weak memory programs without understanding weak memory models.

I give two main results. The first result, presented in Chapter 2, is the local data-race-free guarantee for programmers writing programs that run under weak memory concurrency. It guarantees that reasoning under the simple memory models is still sound in the fully weak memory model under certain conditions. This result was published in PLDI'2021 under the title "Modular Data-Race-Freedom Guarantees in the Promising Semantics" [1].

The other result presented in Chapter 3 is for compiler writers. Surprisingly, we show that reasoning under the sequential memory model is sound even under weak memory concurrency. This result was published in PLDI'2022 under the title "Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency" [2].

The results are formalized and verified against the state-of-the-art weak memory model, the *promising semantics* [3, 4]. The formal explanation and examples of the promising semantics are given in Section 1.2.

1.2 Background: The Promising Semantics

1.2.1 The Promising Semantics

In this section we provide an introduction to the promising semantics. We include only the necessary parts for keeping our presentation self-contained, and refer the reader to [3, 4] for detailed explanations. Our focus is on the version described in [4, §4.4], which we refer to as PS2.1.[1]

We present the fragment of the model containing: *relaxed reads and writes* (`rlx`), *strong relaxed writes* (`srlx`), *release writes* (`rel`), and *acquire reads* (`acq`). Read-

[1]Most of the details, however, are identical for the original PS model and for the PS2 model (the only difference has to do with the notion of "capped memory" and reservations).

				$M, P \subseteq \mathsf{Msg} \cup \mathsf{Rsv}$	memory/promise set
$v \in \mathsf{Val}$	value	$f, t \in \mathsf{Time} \triangleq \mathbb{Q}^+$	timestamp	σ	thread-local program state
$X, Y, Z, L \in \mathsf{Loc}$	location	$(f, t] \in \mathsf{Time} \times \mathsf{Time}$	timestamp interval	$T = \langle \sigma, V, P \rangle \in \mathsf{Lts}$	thread state
$o_\mathtt{R} \in \{\mathtt{rlx}, \mathtt{acq}\}$	read access mode	$V \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$	view	$\langle T, M \rangle$	thread configuration
$o_\mathtt{W} \in \{\mathtt{rlx}, \mathtt{srlx}, \mathtt{rel}\}$	write access mode	$m = \langle X@f, v, t \rangle V \in \mathsf{Msg}$	message	$\mathcal{T} : \mathsf{Tid} \to \mathsf{Lts}$	thread state mapping
$\pi \in \mathsf{Tid} \triangleq \{\pi_1, \pi_2, ...\}$	thread identifier	$r = X@(f, t] \in \mathsf{Rsv}$	reservation	$\langle \mathcal{T}, M \rangle$	machine state

Figure 1.1: Domains and metavariables in PS2.1

(READ-HELPER)
$$\frac{m = \langle X@\_, \_, t \rangle V_\mathtt{m} \in M \qquad V(X) \le t \\ o_\mathtt{R} = \mathtt{rlx} \;\Rightarrow\; V' = V \sqcup [X \mapsto t] \\ o_\mathtt{R} = \mathtt{acq} \;\Rightarrow\; V' = V \sqcup [X \mapsto t] \sqcup V_\mathtt{m}}{\langle V, M \rangle \xrightarrow{o_\mathtt{R}, m}_\mathtt{R} V'}$$

(WRITE-HELPER)
$$\frac{m = \langle X@\_, \_, t \rangle V_\mathtt{m} \qquad V(X) < t \\ o_\mathtt{W} \neq \mathtt{rel} \;\Rightarrow\; V_\mathtt{m} = \bot \\ o_\mathtt{W} = \mathtt{rel} \;\Rightarrow\; (V_\mathtt{m} = V \sqcup [X \mapsto t]) \wedge (P|_X^{\mathsf{Msg}} = \emptyset)}{\langle V, P, M \rangle \xrightarrow{o_\mathtt{W}, m}_\mathtt{W} \langle V \sqcup [X \mapsto t], P, M \uplus \{m\} \rangle}$$

(FULFILL-HELPER)
$$\frac{m = \langle X@\_, \_, t \rangle \bot \in P \qquad V(X) < t \\ o_\mathtt{W} = \mathtt{rlx}}{\langle V, P, M \rangle \xrightarrow{o_\mathtt{W}, m}_\mathtt{W} \langle V \sqcup [X \mapsto t], P \setminus \{m\}, M \rangle}$$

(READ)
$$\frac{\sigma \xrightarrow{\mathtt{R}(o_\mathtt{R}, X, v)} \sigma' \\ m = \langle X@\_, v, \_ \rangle\_ \\ \langle V, M \rangle \xrightarrow{o_\mathtt{R}, m}_\mathtt{R} V'}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{R}(o_\mathtt{R}, m)} \langle \langle \sigma', V', P \rangle, M \rangle}$$

(WRITE)
$$\frac{\sigma \xrightarrow{\mathtt{W}(o_\mathtt{W}, X, v)} \sigma' \\ m = \langle X@\_, v, \_ \rangle\_ \\ \langle V, P, M \rangle \xrightarrow{o_\mathtt{W}, m}_\mathtt{W} \langle V', P', M' \rangle}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{W}(o_\mathtt{W}, m)} \langle \langle \sigma', V', P' \rangle, M' \rangle}$$

(RMW)
$$\frac{\sigma \xrightarrow{\mathtt{RMW}(o_\mathtt{R}, o_\mathtt{W}, X, v_\mathtt{R}, v_\mathtt{W})} \sigma' \\ m_\mathtt{R} = \langle X@\_, v_\mathtt{R}, t \rangle\_ \qquad m_\mathtt{W} = \langle X@t, v_\mathtt{W}, \_ \rangle\_ \\ \langle V, M \rangle \xrightarrow{o_\mathtt{R}, m_\mathtt{R}}_\mathtt{R} V_\mathtt{R} \qquad \langle V_\mathtt{R}, P, M \rangle \xrightarrow{o_\mathtt{W}, m_\mathtt{W}}_\mathtt{W} \langle V', P', M' \rangle}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{RMW}(o_\mathtt{R}, o_\mathtt{W}, m_\mathtt{R}, m_\mathtt{W})} \langle \langle \sigma', V', P' \rangle, M' \rangle}$$

(PROMISE) / (RESERVE)
$$\frac{x \in \mathsf{Msg} \qquad / \qquad x \in \mathsf{Rsv}}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{prm}\,/\,\mathtt{rsv}} \langle \langle \sigma, V, P \uplus \{x\} \rangle, M \uplus \{x\} \rangle}$$

(CANCEL)
$$\frac{r \in P \cap \mathsf{Rsv}}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{cncl}} \langle \langle \sigma, V, P \setminus \{r\} \rangle, M \setminus \{r\} \rangle}$$

(FAIL)
$$\frac{\sigma \xrightarrow{\mathtt{fail}} \bot}{\langle \langle \sigma, V, P \rangle, M \rangle \xrightarrow{\mathtt{fail}} \langle \langle \bot, V, \emptyset \rangle, M \rangle}$$

Figure 1.2: Thread configuration steps in PS2.1

modify-writes (RMWs) carry two access modes—one for the read part and one for the write part. To simplify the presentation, we omit fences and release sequences. We also elide "system calls", used in [3, 4] to specify the observations of a given program. Instead, as we did when analyzing the examples above, we identify behaviors with final outcomes assigning values to certain registers. Nevertheless, our formal development [5] handles *all* features previously included in [3, 4] and uses system calls to define observable behaviors.

Figure 1.1 summarizes the different domains and (implicitly typed) metavariables. To define the **machine states**, besides a set Loc of locations and a set Val of values, we assume a set Time of *timestamps* which are rational numbers (totally and densely) ordered by $<$ with 0 being the minimum value. A *view*, $V \in \mathsf{Loc} \to \mathsf{Time}$, records a timestamp for each location. We represent half-open ranges of timestamps using *timestamp intervals* denoted by $(f, t]$ with $f < t$ or $f = t = 0$. A *machine state* is a pair $\langle \mathcal{T}, M \rangle$, where:

- M, called *memory*, is a finite set of *messages* and *reservations*. A message m

takes the form $\langle X@f, v, t\rangle V$ where: $X \in \mathsf{Loc}$, $(f, t]$ is a timestamp interval (t is called the *timestamp* of m), $v \in \mathsf{Val}$, and $V \in \mathsf{View}$ (called *message view*). In turn, a reservation $r = X@(f, t]$ is defined like a message but without a value and a view. For a memory to be well-formed (as we implicitly assume henceforth), we require that messages/reservations with the same location have disjoint timestamp intervals; and that the view of each message is pointing to a timestamp of an existing message for every location. The initial memory consists of an initialization message $\langle X@0, 0, 0\rangle\bot$ for every location X, where $\bot \triangleq \lambda X.\, 0$ denotes the bottom view.

• $\mathcal{T}$ is a mapping assigning a *thread state* $T = \langle \sigma, V, P\rangle$ to every thread $\pi \in \mathsf{Tid}$, where:

– σ records the (thread-local) *program state.* To keep the presentation abstract, rather than introducing a concrete syntax, we assume that the programming language is represented as a transition system, with local transitions labeled with the action that is performed. Each program state σ consists of the program code, the current program counter and local register file. To run PS2.1 on a program *prog*, we initialize the program state of each thread to include its part of *prog* and the initial program counter and register file.

– V, called the *thread view*, records the highest timestamp that the thread has observed for each location.

– P, called the *thread promise set*, is a set of messages and reservations recording the thread's outstanding promised and reserved writes. Since every promise and reservation is also added to the memory, we will always have $P \subseteq M$.

Importantly, we require thread states to be *well-formed*, where for every location X, the current view of π for X is lower than the timestamp of all of π's outstanding promised writes for X (*i.e.*, $\langle X@\_, \_, t\rangle\_ \in P \Rightarrow V(X) < t$). This condition, called *promise-consistency* in [4], is equivalent to saying that a thread should always be able to fulfill its promises by executing *some* sequence of operations (but not necessarily the sequence dictated by the program).

Figure 1.2 provides the **thread configuration steps**:

read. A thread with view V reads by picking a message $\langle X@f, v, t\rangle V_{\mathtt{m}} \in M$ provided that $V(X) \leq t$, and updating its view for X to t. An acquire read operation incorporates the message view $V_{\mathtt{m}}$ in the thread view (the operator $\sqcup$ "joins" views by taking the pointwise maximum).

write. A thread with view V writes by adding a message m to the memory whose timestamp is greater than the thread's view of X $(V(X) < t)$. Non-release writes set the message view to the bottom view, whereas release writes record the thread view in the message view. Instead of adding a message, relaxed writes may *fulfill* outstanding promises by removing messages from the thread's set of promises. In addition, a release write to a location X forbids the existence of outstanding promises for X (denoted as $P|_X^{\mathsf{Msg}} = \emptyset$).

rmw. A thread performs an RMW by first reading a message $m_{\mathtt{R}} = \langle X@f, v_{\mathtt{R}}, t\rangle V_{\mathtt{R}}$, and then attaching a new message to the read message, *i.e.*, adding a message of the form $m_{\mathtt{W}} = \langle X@t, v_{\mathtt{W}}, t'\rangle V_{\mathtt{W}}$. This results in consecutive messages $(f, t], (t, t']$, forbidding later writes from being placed between the two messages, which guarantees RMW atomicity.

promise. The main novelty of the promising model lies in its way to enable the reordering of a relaxed read followed by a relaxed write (to a different location). It does so by allowing threads to non-deterministically *promise* future (relaxed) writes, by simply adding messages to memory. Outstanding promises are recorded in the thread state, and removed when promises are fulfilled. As described below, to prevent "out-of-thin-air" behaviors (and validate DRF) the outstanding promises at every step are confined by the machine that requires *certification*—a thread that takes a step should always be able to fulfill all its promises when executed in isolation.

reserve. To support register promotion and a more efficient mapping of RMWs to Arm (see Example 2 below), PS2.1 (as well as PS2, but unlike PS) allows threads to reserve timestamp intervals for their own future writes. Unlike promises, reservations do not commit on the *value* that will be used to fill the reserved interval, and thus cannot be read by other threads. They are only used to "block" timestamp intervals in the memory. As in the promise step, a thread adds the reservation to both the memory and its promise set.

cancel. A thread may cancel any of its reservations by simply removing it from the memory and its promise set.

fail. A thread can fail (modeling, *e.g.*, division by 0 or an assertion failure) and invoke UB. Since UB can be replaced by any sequence of actions, this step is considered as fulfilling all of the thread's outstanding promises (here we need the well-formedness assumption on thread states).

The **machine steps** interleave thread configuration transitions as follows:

$$\frac{\langle \mathcal{T}(\pi), M\rangle \xrightarrow{\texttt{cncl}}{}^{*} \xrightarrow{l} \xrightarrow{\texttt{rsv}}{}^{*} \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ is consistent}}{\langle \mathcal{T}, M\rangle \overset{\pi,l}{\Longrightarrow} \langle \mathcal{T}[\pi \mapsto T'], M'\rangle}$$

At each machine step, one thread is performing one thread step, possibly preceded by a sequence of reservation cancellations and followed by a sequence of reservations. Crucially, to ensure that promises do not make the semantics overly weak, a thread cannot take a step unless it reaches a *consistent* configuration, which is defined by:

Definition 1 (Consistency)**.** A thread configuration $\langle T, M\rangle$ is *consistent* if $\langle T, \widehat{M}\rangle \to^* \langle\langle \_, \_, \emptyset\rangle, \_\rangle$ where $\widehat{M}$, called *capped memory*, is the memory obtained from M as follows:

(i) For every message/reservation on X@$(\_, t_1]$ and message/ reservation on X@$(f_2, \_]$ with $t_1 < f_2$, if there is no message/reservation in M with location X and timestamp $t_1 < t < f_2$, add a reservation X@$(t_1, f_2]$; and

(ii) For every message/reservation on $X@(\_, t_{\max}]$ such that there is no message/reservation on $X@(\_, t]$ with $t > t_{\max}$, add a reservation $X@(t_{\max}, t_{\max}+1]$.

Roughly speaking, consistency requires *certification*: the thread that took the step should be able to fulfill all its promises when executed in isolation. The certification starts from a *capped* version $\widehat{M}$ of the current memory M, where all timestamp intervals between existing messages and reservations are blocked by reservations and a "cap reservation" is attached to the message with the highest timestamp for each location. As demonstrated in Example 3 below, a consequence of this is that promises cannot be made across RMW operations. (This is where the PS2.1 and PS2 differ; see [4].)

Behavior A behavior in PS2.1 are defined as follows.[2]

Definition 2. A *behavior* (in PS2.1) is a mapping $r : \mathsf{Tid} \to \mathsf{Val}$ assigning a return value to each thread or $r = \bot$ for erroneous termination. We inductively define when a state $\langle \mathcal{T}, M\rangle$ *generates* a behavior r, denoted by $\langle \mathcal{T}, M\rangle \Downarrow r$:

$$\frac{\forall \pi \in \mathsf{Tid}.\ \ \mathcal{T}(\pi) = \langle \mathbf{return}(v_\pi), \_, \_\rangle}{\langle \mathcal{T}, M\rangle \Downarrow (\lambda \pi. v_\pi)} \qquad \frac{\langle \mathcal{T}(\pi), M\rangle \to^* \langle\langle \bot, V, \emptyset\rangle, M'\rangle}{\langle \mathcal{T}, M\rangle \Downarrow r} \qquad \frac{\langle \mathcal{T}, M\rangle \to \langle \mathcal{T}', M'\rangle \quad \langle \mathcal{T}', M'\rangle \Downarrow r}{\langle \mathcal{T}, M\rangle \Downarrow r}$$

Below, we denote by $[\![prog]\!]_{\mathrm{PS2.1}}$ the set of all behaviors of a program *prog* that are allowed in the PS2.1 semantics.

Remark 1. In [4] the machine step consists of any sequence of thread steps. We observe (and proved in Coq) that by using reservations and cancellations, it is possible to obtain a "normal form" for machine steps: a (possibly empty) sequence of cancellations, followed by a *single* thread step, followed by a (possibly empty) sequence of reservations. This normal form simplifies modular reasoning, as we can assume a consistent state when control is passed between the library code and the client code.

[2] In Coq, a behavior is a sequence of system calls invoked during the program execution. The version in the chapter can be seen as the simplified case where the code of each thread ends with a **return**(e) system call.

Next, we present several instructive examples involving RMWs. We refer the reader to [3, 4] for more examples related to the basic views and promises mechanisms.

Example 1. Two competing RMWs can never read from the same message in memory, as the following annotated program demonstrates:

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

Like **CAS**, we assume that **FADD** returns the value read before the update. Without loss of generality, suppose that π_1 executes first. As it performs an RMW operation, it must "attach" the message it adds to an existing message. Since the only existing message in this stage is the initial one $\langle X@0, 0, 0\rangle\bot$, π_1 will add $m = \langle X@0, 1, t\rangle\bot$ with some $t > 0$ to the memory. Then, the RMW of π_2 cannot also read from the initial message because this would require π_2's message to be attached to the initial message, which would overlap with the $(0, t]$ interval of m.

Example 2. The following annotated program illustrates a drawback of the original PS that prevents register promotion and the intended mapping to Armv8 [6]:

$$\left.\begin{array}{l} a := X^{\texttt{rlx}} \quad /\!/ 1 \\ b := \mathbf{FADD}^{\texttt{acqrel}}(Z, 1) \quad /\!/ 0 \\ Y^{\texttt{rlx}} := 1 \end{array}\;\right\|\; \begin{array}{l} c := Y^{\texttt{rlx}} \quad /\!/ 1 \\ X^{\texttt{rlx}} := c \end{array} \tag{Arm-weak}$$

The annotated behavior is allowed by Armv8 (for the compiled program), and can be also obtained if the thread-local location Z is made a register. It is, however, disallowed by PS. PS2 and PS2.1 solve this problem using *reservations*. To observe $a = 1$, π_1 should be able to promise the write of 1 to Y at the beginning of the execution. This is not possible without reservations because π_1 cannot update Z during the certification against the capped memory. However, π_1 can *reserve* the interval $(0, 1]$ for the **FADD** before making the promise $Y = 1$. Then, it can certify the promise $Y = 1$ by using the reserved interval to perform the **FADD**. Intuitively speaking, while PS2.1 forbids the reordering of an RMW followed by a store, using reservations, it enables the reordering of the read part of the RMW before the read of X and the write part of the RMW after the write of Y, which more faithfully captures Arm's load-linked/store-conditional implementation.

Example 3. The following annotated program shows a behavior forbidden by PS2.1 because of its stronger certification requirement w.r.t. PS and PS2.

$$\left.\begin{array}{l} a := \mathbf{FADD}(X, 1) \quad /\!/ 1 \\ Y^{\texttt{rlx}} := 1 \end{array}\;\right\|\; \begin{array}{l} b := Y^{\texttt{rlx}} \\ c := \mathbf{FADD}(X, b) \end{array} \tag{RMW-W}$$

For π_1 to read 1 via its **FADD**, it has to promise $Y = 1$. Unlike PS and PS2, this is not allowed in PS2.1 because π_1 cannot perform **FADD** to X during the certification against the capped memory. Promising the **FADD** or reserving a space for it by π_1 is impossible as well. Once π_1 promises its **FADD**, it is committed to update X from 0 to 1. If π_1 reserves a timestamp interval $(0, t]$ for its **FADD**, π_2 cannot update X from 0 to 1 since the $X = 0$ message is blocked by π_1's reservation, again forcing π_1 to update X from 0 to 1.

1.2.2 Additional Examples of the Promising Semantics

We present several instructive examples to assist the reader in understanding the promising model.

Example 4. Consider the simple store-buffering litmus test:

$$\begin{array}{l||l} X^{\texttt{rlx}} := 42 & Y^{\texttt{rlx}} := 37 \\ a := Y^{\texttt{rlx}} \quad /\!/\, 0 & b := X^{\texttt{rlx}} \quad /\!/\, 0 \end{array} \tag{SB}$$

In this example, both threads are allowed to read 0 from the initialization messages. When π_1 performs the write to X, it will add a message $\langle X@f, 42, t\rangle\bot$ by choosing some $t > f \geq 0$. During this write, π_1 should increase its view of X to t, while maintaining $V(Y)$ to be 0 as it was. Hence, π_1 is still allowed to read 0 from Y in the subsequent execution. As π_2 can be executed in the same way, both threads are allowed to read 0.

Example 5. To see how release/acquire synchronization works consider the simple message passing litmus test:

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

Here, if π_2 reads 1 from Y, which is written by π_1, both threads are synchronized through release and acquire. Thus, π_2 obtains the knowledge of π_1, namely its view for X is increased to include the timestamp of $X^{\texttt{rlx}} := 1$ of π_1. Therefore, after reading 1 from Y, π_2 is not allowed to read the initial value from X.

Example 6. To see how promises work consider the simple load buffering litmus test:

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

To obtain the annotated behavior (which has to be allowed in a model supporting load-store reordering of relaxed accesses), π_1 may promise $Y^{\texttt{rlx}} := 1$ at first. This allows π_2 to read 1 from Y and write it back to X. Then, π_1 can read 1 from X, which was written by π_2, and fulfill its promise. Note that at the point of promising $Y^{\texttt{rlx}} := 1$ (in the very beginning of the run), π_1 can run and perform $Y^{\texttt{rlx}} := 1$ without any "help" of other threads, so it reaches a consistent thread state after making the promise.

Example 7. Certification, the thread-local run fulfilling all outstanding promises of the thread, is necessary to avoid "thin-air reads" as demonstrated by the following variant of LB:

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

As every thread simply copies the value it reads, both threads are not supposed to read any non-0 value. In the promising semantics, if a thread could promise without certification, this behavior would be allowed by the same execution as the one for LB above. However, with the certification requirement, π_1 cannot promise $Y^{\texttt{rlx}} := 1$, as, when running in isolation, π_1 only writes $Y^{\texttt{rlx}} := 0$.

Chapter 2

Local Data-Race-Freedom Guarantees for Program Writers

2.1 Introduction: The Need for Local DRF

Designing a programming language shared-memory concurrency semantics, a.k.a. a weak memory model, is a complex task. On the one hand, one aims to allow mappings to commodity modern architectures (such as x86, Power, Arm, and RISC-V) that will not subvert the hardware's extensive optimization efforts, as well as to validate certain compiler optimizations that are unsound under a strong semantics such as sequential consistency (SC). On the other hand, since the introduction of weak memory semantics in programming languages, it was clear that the majority of programmers will need to program and reason about their code without understanding the full complexities of the underlying semantics. Hence, to be useful and amenable to reasoning, a memory model has to (i) ensure strong and intuitive semantics for programs that follow certain programming disciplines; and (ii) allow programmers to adhere to such disciplines even without knowing the actual underlying weak semantics.

A fundamental programmability guarantee of this kind is DRF-SC [7]. It ensures that data-race free programs (avoiding races using locks or designated synchronization accesses) only exhibit SC behaviors. Crucially, data-race freedom (DRF), the premise of DRF-SC, is only required to hold *under SC*, allowing programmers to use this guarantee knowing nothing about the underlying complex model, but rather naively imagining standard interleaving semantics that follows the program order and employs a conventional memory.

Since SC is sometimes considered overly expensive to ensure efficient implementations (and as building blocks for establishing DRF-SC), more refined DRF guarantees have been studied in the last few years [3, 8]. Each of these guarantees is applicable on a different level of accesses—requiring more restrictive race-freedom conditions and resulting in stronger semantics guarantees. In particular, in models with release/acquire (RA) accesses, one aims to ensure RA semantics for programs that exhibit no races on accesses weaker than RA accesses. This guarantee, called DRF-RA [3], allows programmers to use (non-racy) weaker (and more efficient) accesses than RA accesses while knowing *only* the RA semantics. The latter, although weaker than SC, is much simpler than the full underlying model, and it admits several verification methods and tools, including model checkers and program logics [9, 10, 11, 12]. Similarly, on the level of "relaxed" accesses, which are weaker than RA ones and intended to be compiled to plain machine loads and stores, a DRF guarantee with respect to an "in-order" RC11-like [13] semantics (with no load buffering behaviors) ensures in-order semantics when, under the in-order semantics, races on relaxed accesses are properly confined (see DRF-PF in [3] and DRF-RLX in [8]). Again, the benefit is significant: an in-order semantics like RC11 is significantly simpler than an "out-of-order" model in which reads can read from later writes, and like SC and RA, "in-order" models admit several verification methods [14, 15, 16, 17].

Nevertheless, the global nature of all DRF guarantees mentioned above makes them only applicable when the whole program admits the required race freedom premise. Software, however, is modularly developed, often without access to the full code. Moreover, benign races in carefully crafted concurrency libraries make the DRF guarantees futile for reasoning by clients that use these libraries, leaving them with no formal assurances applicable without a complete understanding of the underlying model (see Fig. 2.1 for an illustrative example).

This drawback of the DRF guarantees is addressed by more refined "local" guarantees that can be applied also on *parts* of a given program [18, 19]. In particular, a local DRF (LDRF, for short) guarantee allows one to conclude that accesses to *certain shared locations* have stronger semantics provided that when assuming stronger semantics *to these locations*, the program exhibits no races *on them*. The important practical consequence is that it is safe to assume that the client portion of the code is running under the stronger semantics when races are completely confined in the library code. Moreover, clients may rely on the synchronization guarantees provided by libraries to establish race freedom of their code while still understanding only the stronger semantics.

Unfortunately, the negative observation of this chapter is that LDRF guarantees of this kind are inconsistent with compiler optimizations that are normally expected to be sound in weak memory models. To demonstrate this, we present examples that, under very minimal assumptions on the underlying model, are locally race free, but a sequence of program transformations that are intended to be sound entails that they must have a behavior that violates LDRF. Viewing these guarantees as essential for modular software development, we believe that this reveals a severe limitation on the usefulness of models that support the full range of optimizations.

On the positive side, we observe that by disabling a certain problematic compiler optimization, an LDRF guarantee w.r.t. an "in-order" RC11-like semantics becomes achievable. Concretely, we identify that *RMW-store reordering*[1] is the source of the problem and show that by disabling only these reorderings one is able to validate a critical LDRF guarantee. In turn, for (naive formulations of) LDRF-RA/SC, disabling RMW-store reordering does not suffice, and we address the problem by slightly strengthening the (naive) race-freedom premise. The resulting guarantees are useful for modular reasoning (as demonstrated in §2.4), and we are not aware of any non-contrived example where this strengthened race-freedom condition makes a difference.

To establish that forbidding RMW-store reordering and slightly strengthening the race-freedom premise suffice for establishing the LDRF guarantees, we demonstrate a particular model that satisfies the desiderata. Concretely, we prove that three LDRF guarantees are validated by PS2.1, a variant of the promising semantics, mentioned as a possible simplification of PS2 in [4, §4.4], that supports all standard (local and global) optimizations excluding RMW-store reordering.

In addition to the theoretical results, we empirically investigate the cost of forbidding RMW-store reorderings, and observe that it is negligible in practice. Current standard compilers are very conservative with reorderings of atomic accesses [21], and mainstream architectures, except Armv8, do not allow RMW-store reordering. Even in Armv8, it is relevant only for non-acquire *unconditional* RMWs (*i.e.*, **FADD** or **XCHG**, but not **CAS**), for which a "fake" branch instruction is needed to prevent the reordering. Since **FADD**s and **XCHG**s are not executed frequently and fake branching is relatively cheap [21], we expect the implementation cost to be negligible in practice. We have performed a sequence of experiments that validate this hypothesis (§2.5).

As for the LDRF guarantees, we formulate three guarantees, and prove them

[1]RMW (read-modify-write) operations, such as compare-and-swap (**CAS**), fetch-and-add (**FADD**) and atomic exchange (**XCHG**), *atomically* perform a read followed by a write to the same location. Certain models—*e.g.*, C11 [20], the promising semantics [3], and Weakestmo [8]—allow the reordering of non-acquire RMWs with subsequent relaxed writes to a different location.

```
$r_0$ := pop_wait($S$)              ║ $r_1$ := pop_wait($S$)
lock()                             ║ lock()
  process $r_0$ accessing $X$, $Y$  ║   process $r_1$ accessing $X$, $Y$
unlock()                           ║ unlock()
```

Two threads are popping "work items" from a wait-free (possibly, relaxed) stack S, and use a lock to perform the work for avoiding races on shared locations X and Y. The DRF-SC guarantee does not allow the client to show that the accesses to X and Y inside the locked regions do not have weak behaviors. Indeed, the program is not race free due to benign races in the implementation of the pop operation (in fact, if lock/unlock are not primitives, then the implementation of the lock itself is racy as well). In contrast, a *local* DRF-SC guarantee allows clients to use the specification of the lock to conclude that the accesses to X and Y are not racy, and therefore, they can safely assume SC semantics for X and Y.

Figure 2.1: A simple example demonstrating the weakness of the global DRF-SC guarantee

for PS2.1, where each of which provides the key lemma for establishing the next one: (1) LDRF-PF w.r.t. the *promise-free* (RC11-like) semantics allowing one to restrict "promises"—a special mechanism that accounts for load-store reorderings in the promising semantics, which is undoubtedly the most complicated and hard to reason about component of the model; (2) LDRF-RA w.r.t. release/acquire semantics; and (3) LDRF-SC w.r.t. SC semantics.[2]

To conclude, our contributions are summarized as follows:

1. We show that the full set of compiler optimizations is inconsistent with local DRF guarantees (§2.2).
2. We establish the consistency of three local DRF guarantees (LDRF-PF, LDRF-RA,

[2]Although allowing races on SC accesses is essentially needed for *global* DRF-SC (otherwise there are no means of synchronization), it is unnecessary for *local* DRF-SC because synchronization is typically provided by library methods. Thus, LDRF-SC is still applicable for the promising semantics, which currently lacks specialized SC accesses.

and LDRF-SC) and all standard optimizations excluding RMW-store reordering by proving that PS2.1 validates them all (§2.3).

3. We outline the applicability of local DRF for reasoning about client code, as well as library code (§2.4).
4. We empirically observe that the performance impact of disabling RMW-store reorderings is negligible (§2.5).

Our LDRF proofs in §2.3 are fully mechanized in Coq. The formalization is available at [5].

2.2 Local DRF in Weak Memory Models

In this section we demonstrate the inherent tension between local DRF guarantees and standard compiler optimizations. While our results in the next sections are specific to the promising semantics, the discussion in this section is general, making its implications applicable in other models as well.

2.2.1 Local DRF w.r.t. an "In-Order" Semantics

By far, the most complicated aspect of a weak memory semantics is related to allowing load-store reordering of possibly racy independent relaxed accesses (a.k.a. load buffering behaviors). This is the source of the infamous "out-of-thin-air" problem [22], the reason why per-execution declarative models cannot work and more complicated event-structure-based models are needed instead [23, 8], and the only rationale behind "promises" in the promising semantics. To circumvent this complexity, one can use less efficient stronger models, such as RC11 [13], that conservatively forbid load-store reorderings altogether (by disallowing cycles in the union of the program order and the reads-from relation), and thus cannot map relaxed accesses to plain machine loads and stores in architectures like Arm.

The compiler may optimize Thread 1 as shown below:

(0)
```
$Y := 0$


$a := Y^{\texttt{rlx}}$
if $a \neq 0$ then
  $b :=$ CAS$(X, 0, 42)$
  if $b = 0$ then
    $c := L^{\texttt{rlx}}$
    if $c = 1$ then
      $X^{\texttt{srlx}} := 37$
```

(1)
```
$Y := 0$
$c := L^{\texttt{rlx}}$

$a := Y^{\texttt{rlx}}$
if $a \neq 0$ then
  $b :=$ CAS$(X, 0, 42)$
  if $b = 0$ then

    if $c = 1$ then
      $X^{\texttt{srlx}} := 37$
```

(2)
```
$Y := 0$
$c := L^{\texttt{rlx}}$
if $c = 1$ then
  $a := Y^{\texttt{rlx}}$ else $a := Y^{\texttt{rlx}}$
if $a \neq 0$ then
  $b :=$ CAS$(X, 0, 42)$
  if $b = 0$ then

    if $c = 1$ then
      $X^{\texttt{srlx}} := 37$
```

(3)
```
$Y := 0$
$c := L^{\texttt{rlx}}$
if $c = 1$ then
  $a := Y^{\texttt{rlx}}$ else $a := 0$
if $a \neq 0$ then
  $b :=$ CAS$(X, 0, 42)$
  if $b = 0$ then

    if $c = 1$ then
      $X^{\texttt{srlx}} := 37$
```

(4)
```
$Y := 0$
$c := L^{\texttt{rlx}}$
if $c = 1$ then
  $a := Y^{\texttt{rlx}}$
  if $a \neq 0$ then
    $b :=$ CAS$(X, 0, 42)$
    if $b = 0$ then

      (eliminated)
      $X^{\texttt{srlx}} := 37$
else $a := 0$
    (eliminated)
```

(5)
```
$Y := 0$
$c := L^{\texttt{rlx}}$
if $c = 1$ then
  $a := Y^{\texttt{rlx}}$
  if $a \neq 0$ then
    $b :=$ CAS$^{\texttt{srlx}}(X, 0, 37)$



else $a := 0$
```

(1) reorder the read $c := L$ to be second, after introducing the same read $c := L$ in the else-branches (when $b \neq 0$ or $a = 0$);

(2) insert a dummy if-then-else on $c = 1$ and distribute the rest of the code to both branches ("trace-preserving" transformation);

(3) forward the write $Y := 0$ to the read $a := Y^{\texttt{rlx}}$ in the else-branch, turning it into $a := 0$;

(4) distribute the branch on $a \neq 0$ to both prior branches on $c = 1$ and optimize them: eliminate repeated redundant testing of $c = 1$ in the then-branch, and remove dead code in the else-branch ("trace-preserving" transformation);

(5) merge $b :=$ **CAS**$(X, 0, 42)$ and **if** $b = 0$ **then** $X^{\texttt{srlx}} := 37$ into $b :=$ **CAS**$^{\texttt{srlx}}(X, 0, 37)$.

Now, the compiler may optimize Thread 2 as shown below:

(0)
```
$Y := 1$
$d :=$ CAS$(X, 0, 1)$
if $d \neq 42$ then
  $L^{\texttt{rlx}} := 1$
```

(1)
```
$Y := 1$
$d :=$ CAS$(X, 0, 1)$
(eliminated)
$L^{\texttt{rlx}} := 1$
```

(2)
```
$Y := 1$
$L^{\texttt{rlx}} := 1$

$d :=$ CAS$(X, 0, 1)$
```

(1) noticing that $X \neq 42$ is a global invariant (42 is never written to X), optimize away the redundant test "**if** $(d \neq 42)$ **then**";

(2) reorder the independent **CAS** on X and write to L.

Figure 2.2: Program transformations on LDRF-PF-Fail (in the final transformed program, we may get $d = 37$ even under SC!)

We generally refer to RC11-like models as "in-order" models, as they are captured by transition systems that execute memory accesses according to their program order while ensuring that every read reads from a previously executed write. More formally, this property is defined as follows:

Definition 3. A memory model M is *in-order* if every behavior allowed by M corresponds to a trace of memory accesses that respects the program order such that every read r of value v from location X is justified by some write w that writes v to X and appears in the trace *before* r.

This definition covers a wide variety of (not so weak) memory models including RC11, TSO [24], causal consistency [25, 26], the OCaml model in [18], and (of course) SC. It ensures a conceptually simple semantics and enables several verification approaches [14, 15, 16, 17].

A natural approach to allow "in-order" reasoning for a given program in a model with (fully) relaxed accesses is to use a DRF guarantee. When such guarantee is provided, one is able to assume in-order semantics for programs that under in-order semantics exhibit no races on accesses annotated as relaxed (so that the guarantee can be applied knowing nothing about the out-of-order part of the semantics). Moreover, as demonstrated in §2.1, for being applicable in a modular fashion (*e.g.*, in the presence of unrelated races induced by some library methods over which the client has no control), this guarantee has to be local.

To give a more precise statement of such a local DRF guarantee (but still keep the discussion general), consider an arbitrary model M with relaxed reads/writes, intended to be compiled to plain machine accesses, and "strong relaxed" writes, intended to be compiled with barriers to forbid the hardware from reordering a load followed by a strong relaxed write.[3] Strong relaxed writes provide "in-order" semantics in the

[3] Strong relaxed accesses were introduced in [3] as a technical tool for establishing the correctness of mapping to hardware. They are also useful in the current discussion. Like release writes, they forbid reordering with preceding reads; but unlike release writes, they are not intended to synchronize with reads by other threads.

following sense: Every behavior allowed by M corresponds to some trace of memory accesses that respects the program order such that: (i) every read r of value v from location X is justified by some write w that writes v to X and appears in the trace; and (ii) if r is justified by w that is strong relaxed, then w should appear before r in the trace. (Note that M allows a read r to be justified by a *relaxed* write that is executed after r.)

Then, a local DRF guarantee w.r.t. an in-order semantics for M is stated as follows: For every set $\mathcal{L}$ of locations, every behavior of a given program *prog* allowed by M is allowed by M for *prog* when all writes to locations in $\mathcal{L}$ are considered strong relaxed, provided that under this assumption *prog* exhibits no races involving writes to locations in $\mathcal{L}$ that are annotated as relaxed.

For example, this guarantee (for $\mathcal{L} = \{L\}$) allows one to show that the annotated behavior in the following program is disallowed in the model M without knowing anything besides an in-order semantics:[4]

$$\begin{array}{l||l} a := L^{\mathtt{rlx}} & b := X^{\mathtt{rlx}} \quad /\!\!/ \mathit{1\ ?} \\ \mathbf{libfun_1}() & \mathbf{libfun_2}() \\ X^{\mathtt{srlx}} := a & \textbf{if } b = 1 \textbf{ then } L^{\mathtt{rlx}} := 1 \textbf{ else } L^{\mathtt{srlx}} := 1 \end{array} \qquad \text{(LDRF-LB)}$$

where $\mathbf{libfun_1}()$ and $\mathbf{libfun_2}()$ are calls to some library methods that execute racy relaxed code accessing a set of locations disjoint from X and L. Indeed, assuming that $L^{\mathtt{rlx}} := 1$ has strong relaxed semantics, all writes to X and L are strong relaxed, and the in-order property easily entails that $L^{\mathtt{rlx}} := 1$ (in the then-branch) is never executed and thus not involved in a race. Then, the premise of the LDRF guarantee above holds, and one concludes, again based on the in-order property, that $b = 1$ is disallowed by M. Crucially, this reasoning does not require any knowledge of how exactly M behaves for (fully) relaxed writes (which, in fact, we have not specified). We also note that a global DRF guarantee cannot be used due to the presence of racy

[4]We assume that all locations are initially 0, and that the “default” access mode is relaxed (so we omit `rlx` annotations).

code in the library methods.

Unfortunately, we observe that this LDRF guarantee is actually inconsistent with program optimizations that are standardly intended to be sound in weak memory models. Indeed, the following example shows that any such model M cannot validate both the LDRF guarantee and all standard optimizations:[5]

```
$Y := 0$                          ||
$a := Y^{\texttt{rlx}}$                  ||
**if** $a \neq 0$ **then**                 || $Y := 1$
   $b := \textbf{CAS}(X, 0, 42)$        || $d := \textbf{CAS}(X, 0, 1)$  $/\!/$ *37 ?*      (LDRF-PF-Fail)
   **if** $b = 0$ **then**                || **if** $d \neq 42$ **then**
      $c := L^{\texttt{rlx}}$             ||    $L^{\texttt{rlx}} := 1$
      **if** $c = 1$ **then**             ||
         $X^{\texttt{srlx}} := 37$        ||
```

where $\textbf{CAS}(X, v_1, v_2)$ reads a value from X; if it is equal to v_1 (*i.e.*, successful), writes v_2 to X ensuring atomicity between the read and write, and otherwise (*i.e.*, unsuccessful) does nothing; and finally returns the read value.

Indeed, assuming that the write to L has strong relaxed semantics, it is easy to see that no execution of the program executes both $c := L^{\texttt{rlx}}$ and $L^{\texttt{rlx}} := 1$, and hence there is no race on L. Specifically, if such execution is allowed by the model M, then the **CAS** of the second thread must read 37 due to the standard RMW atomicity (which implies that two successful CAS instructions cannot read from the same write), and so $c := L^{\texttt{rlx}}$ must read 1. However, since 37 is written by a strong relaxed write, it follows that $X^{\texttt{srlx}} := 37$ appears in the trace before $d := \textbf{CAS}(X, 0, 1)$. This implies that $c := L^{\texttt{rlx}}$ appears in the trace before $L^{\texttt{rlx}} := 1$, which contradicts the assumption that $L^{\texttt{rlx}} := 1$ has strong relaxed semantics.

Now we can demonstrate the inconsistency between the local DRF guarantee above and program optimizations. Since the premise of the guarantee is satisfied (for $\mathcal{L} = \{L\}$), if the guarantee holds, we may assume that $L^{\texttt{rlx}} := 1$ has strong relaxed

[5]We are not aware of a smaller example that can be used for this purpose.

semantics, and conclude, by the exact same reasoning, that the $d = 37$ outcome is disallowed (no execution executes both $c := L^{\texttt{rlx}}$ and $L^{\texttt{rlx}} := 1$). Nevertheless, Fig. 2.2 shows that starting from this program, a sequence of transformations, each of which is intended to be sound in standard weak memory models, may actually lead to the $d = 37$ outcome!

As a concrete example for a model M, consider the PS2 model [4], which satisfies the above assumptions and validates all transformations used in Fig. 2.2. (In PS2, strong relaxed writes correspond to relaxed writes that cannot be promised ahead of their execution.) It follows that PS2 fails to admit the above guarantee w.r.t. an in-order semantics.[6]

To locate the source of the problem, we observe that RMW-store reordering (applied in the second thread's code in Fig. 2.2) is the transformation that breaks a key property, which we call *promise monotonicity* (formally stated in §2.3.1), needed for our proof. Indeed, one of the main ideas in proving local DRF is to show that relaxed store hoisting (moving a relaxed write to be before other instructions) does not allow more behaviors unless the store was racy before the code motion. However, this property fails if reordering of a relaxed RMW followed by a relaxed write to a different location is allowed. For instance, in the program above, executing $d := \mathbf{CAS}(X, 0, 1)$ before $L^{\texttt{rlx}} := 1$ prevents the behavior executing both $L^{\texttt{rlx}} := 1$ and $c := L^{\texttt{rlx}}$, but executing them in the opposite order allows that behavior.

Accordingly, to accomplish our proof, we switched to PS2.1, a variant of the PS2 model outlined in [4, §4.4], which gives up RMW-store reordering for simplicity and better meta-theoretic properties such as the absence of deadlocking executions.[7] For PS2.1 we are able to prove LDRF-PF—a local DRF guarantee with respect

[6]The original promising model PS [3] does not admit global value-range analysis, which is needed in the sequence of transformations in Fig. 2.2. Nevertheless, in §2.6, we present a similar (yet more intricate) counterexample for PS.

[7]We have formally established the absence of deadlocks in Coq [5].

to the promise-free fragment of the promising semantics (an in-order model), thus establishing the consistency of such a local DRF guarantee with all optimizations except for RMW-store reordering.

2.2.2 Local DRF w.r.t. RA and SC

For less advanced users, an in-order RC11-like semantics may still be hard to reason with. Then, one needs local DRF properties w.r.t. stronger fragments like release/acquire semantics (LDRF-RA) or even sequential consistency (LDRF-SC). Next, we discuss the subtlety in stating and achieving these local guarantees in a general model that supports load-store reordering of relaxed accesses. We focus on LDRF-RA, but the discussion is the same for LDRF-SC.

A naive notion of LDRF-RA can be naturally derived from the global DRF-RA guarantee. The latter ensures that a program has only RA behaviors provided that under RA semantics it exhibits no races involving accesses annotated as (strong) relaxed [3]. To "localize" this guarantee with respect to a given set $\mathcal{L}$ of locations, we need to consider "$\mathcal{L}$-RA behaviors"—behaviors in which accesses to locations in $\mathcal{L}$ are treated as RA accesses (even when annotated with weaker modes), but other accesses are interpreted as annotated in the program. Then, a naive LDRF-RA guarantee would say that a program has only $\mathcal{L}$-RA behaviors provided that its $\mathcal{L}$-RA behaviors exhibit no races involving accesses to locations in $\mathcal{L}$ annotated with access modes weaker than release and acquire.

We show that in any sensible weak memory model this guarantee is actually inconsistent with standard program optimizations (here, RMWs are not involved at all).

Specifically, the $\{L\}$-RA behaviors of the program on the right exhibits no races on the location L, but a sequence of standard optimizations may lead to a non $\{L\}$-RA behavior, which invalidates the naive LDRF-RA guarantee.

```
$a := Y^{\texttt{rlx}}$  $/\!/ 1\ ?$          ||
if $a = 1$ then                               ||  $c := X^{\texttt{rlx}}$
    $b := L^{\texttt{rlx}}$                   ||  $L^{\texttt{rlx}} := 1$
    $X^{\texttt{rlx}} := b$                   ||  $Y^{\texttt{rlx}} := c$
else                                          ||
    $X^{\texttt{rlx}} := 1$                   ||
```

(Naive-LDRF-RA-Fail)

To see this, we first claim that in any sensible model, assuming that the accesses to L are RA, the first thread cannot read 1 from Y. Indeed, if $a := Y^{\texttt{rlx}}$ reads 1, it easily follows that $b := L^{\texttt{rlx}}$ reads from $L^{\texttt{rlx}} := 1$. However, with the assumption that the accesses to L are interpreted as RA accesses, the latter implies a "happens-before" path from $c := X^{\texttt{rlx}}$ to $X^{\texttt{rlx}} := b$, which implies that $c := X^{\texttt{rlx}}$ cannot read from $X^{\texttt{rlx}} := b$. In turn, the value 1 is never written to Y.

Second, with the same assumption (that the accesses to L are RA), the above reasoning also shows that there are no races on L. In fact, the exact definition of a race does not matter here: we actually know that the first thread will not access L at all. Then, the naive LDRF-RA for $\mathcal{L} = \{L\}$ implies that the program has only $\{L\}$-RA behaviors, which, as argued above, entails that the $a = 1$ outcome is disallowed. Nevertheless, in Fig. 2.3, we show that starting from the above program, a sequence of program transformations, each of which is intended to be sound in standard weak memory models, may actually lead to this outcome!

What went wrong? In the analysis above, we used a racy access (to L) to establish synchronization, and then used this synchronization to invalidate the racy execution itself. However, when the racy read is performed as a relaxed read it does not induce synchronization, and nothing actually forbids the candidate racy execution. To solve this problem, we have to strengthen the premise of LDRF-RA, so that synchronization induced by racy reads (from locations in $\mathcal{L}$) cannot be used to eliminate races.

A possible way to do so is to weaken the semantics of "racy reads" from locations

The compiler may optimize Thread 1 as shown below:

(0)	(1)	(2)	(3)	(4)
	$b := L^{\texttt{rlx}}$	$b := L^{\texttt{rlx}}$	$b := L^{\texttt{rlx}}$	$b := L^{\texttt{rlx}}$
		if $b = 1$ **then**	**if** $b = 1$ **then**	**if** $b = 1$ **then**
$a := Y^{\texttt{rlx}}$	$a := Y^{\texttt{rlx}}$	$a := Y^{\texttt{rlx}}$	$a := Y^{\texttt{rlx}}$	$X^{\texttt{rlx}} := 1$
if $a = 1$ **then**	**if** $a = 1$ **then**	**if** $a = 1$ **then**	$X^{\texttt{rlx}} := 1$	$a := Y^{\texttt{rlx}}$
$b := L^{\texttt{rlx}}$				**else** ...
$X^{\texttt{rlx}} := b$	$X^{\texttt{rlx}} := b$	$X^{\texttt{rlx}} := b$		
else	**else**	**else**		
$X^{\texttt{rlx}} := 1$	$X^{\texttt{rlx}} := 1$	$X^{\texttt{rlx}} := 1$		
		else ...	**else** ...	

(1) reorder the read $b := L$ to be first, after introducing the same read $b := L$ in the else-branch (when $a \neq 1$);

(2) insert a dummy if-then-else on $b = 1$ and distribute the rest of the code to both branches ("trace-preserving" transformation);

(3) in the then-branch on $b = 1$, substitute b with 1 and merge both branches on $a = 1$ ("trace-preserving" transformation);

(4) reorder the independent read from Y and write to X.

In addition, the compiler may optimize Thread 2 as shown below:

(0)	(1)
$c := X^{\texttt{rlx}}$	$L^{\texttt{rlx}} := 1$
$L^{\texttt{rlx}} := 1$	$c := X^{\texttt{rlx}}$
$Y^{\texttt{rlx}} := c$	$Y^{\texttt{rlx}} := c$

(1) reorder $c := X^{\texttt{rlx}}$ and $L^{\texttt{rlx}} := 1$.

Figure 2.3: Program transformations on Naive-LDRF-RA-Fail (after the transformations, we may get $a = 1$ even under SC!)

in $\mathcal{L}$ in the $\mathcal{L}$-RA semantics, and say that unlike standard acquire reads, these reads do not induce synchronization. However, this solution would require a precise definition of the semantics of racy reads, which goes beyond standard RA semantics.

In this paper, we follow an alternative approach that involves a certain over-approximation—we will say that a racy read simply invokes undefined behavior (UB). Since UB includes any possible behavior, the race-freedom condition based on racy reads invoking UB implies the one where racy reads do not induce synchronization. In other words, we will say that a race occurs if some racy read is reachable ignoring what happens after the racy read is executed. With this definition, relying on the previously mentioned LDRF-PF as a key lemma, we proved LDRF-RA (and LDRF-SC) for PS2.1.[8] Importantly, unlike C11's "catch-fire" semantics, UB for races is not a part of the concurrency semantics (indeed, the promising semantics provides means to avoid "catch-fire"), but is only used for defining races when establishing the premise of LDRF-RA/SC. We note that a similar strengthening of the race-freedom premise in LDRF-PF does not solve the problem outlined in §2.2.1 (LDRF-PF-Fail is still a counterexample).

2.3 Local DRF Guarantees

In this section we present our local DRF results for PS2.1.

We note that, unlike the conventional DRF theorems, write-write races are only considered as races for LDRF-SC. The other results, LDRF-PF and LDRF-RA only require the absence of certain read-write races.

We present the statements of "time-wise" local DRF guarantees in §2.3.4. Roughly speaking, these time-wise guarantees apply when no race occurs between two states in the machine trace and they ensure the stronger semantics between these two states.

[8]PS2 does not satisfy our LDRF-RA/SC theorems (LDRF-PF-Fail is a counterexample for them as well).

All results of this section (including time-wise LDRF) are fully mechanized in Coq ($\sim 35K$ LoC altogether) [5].

2.3.1 Local DRF-PF

The first step for formulating LDRF-PF is to formally define an "in-order" restriction of PS2.1 w.r.t. a given set $\mathcal{L}$ of locations. This can be simply defined by forbidding promises to the locations in $\mathcal{L}$.

Definition 4. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-PF-machine* is the strengthening of PS2.1 obtained by forbidding the application of the (PROMISE) rule for locations in $\mathcal{L}$. We denote by $[\![prog]\!]^{\mathcal{L}}_{\text{PF}}$ the set of all behaviors of a program *prog* allowed by the $\mathcal{L}$-PF-machine.

Next, we define what a racy execution in the $\mathcal{L}$-PF-machine is. Roughly, an execution is $\mathcal{L}$-racy if it includes some thread π_1 taking a machine step writing a message m to a location in $\mathcal{L}$ by a relaxed write, immediately followed by another thread π_2 taking a sequence of machine steps that ends with reading the message m.

Definition 5. An execution in the $\mathcal{L}$-PF-machine is *$\mathcal{L}$-racy* if it includes a sequence of machine steps of the form:

$$\xrightarrow{\pi_1, l_1} \xrightarrow{\pi_2, \_}{}^{*} \xrightarrow{\pi_2, l_2}$$

with $\pi_1 \neq \pi_2$, $l_1 \in \{\mathtt{W}(\mathtt{rlx}, m), \mathtt{RMW}(\_, \mathtt{rlx}, \_, m)\}$ and $l_2 \in \{\mathtt{R}(\_, m), \mathtt{RMW}(\_, \_, m, \_)\}$ for $m \in \mathsf{Msg}$ with location $L \in \mathcal{L}$.

Then, LDRF-PF is formulated as follows.

Theorem 1 (LDRF-PF)**.** If there is no $\mathcal{L}$-racy execution of *prog* in the $\mathcal{L}$-PF-machine, then $[\![prog]\!]_{\text{PS2.1}} = [\![prog]\!]^{\mathcal{L}}_{\text{PF}}$.

Remark 2. While the $\mathcal{L}$-PF-machine forbids promises to locations in $\mathcal{L}$, it still allows making reservations to these locations. Nevertheless, the $\mathcal{L}$-PF-machine is an in-order semantics w.r.t. $\mathcal{L}$ since threads cannot read from reservations. Moreover, the only purpose of making reservations to $\mathcal{L}$ is to allow certain promises to locations not in $\mathcal{L}$. Hence, reservations to $\mathcal{L}$ can be ignored in the typical use of LDRF that over-approximates the behaviors of locations not in $\mathcal{L}$ to be completely unconstrained.

Revisiting LDRF-PF-Fail, the argument outlined in §2.2 shows that no execution of LDRF-PF-Fail in the $\{L\}$-PF-machine is L-racy. Then, from Thm. 1, it follows that the $d = 37$ outcome is disallowed for that program under PS2.1.

Example 8. As an instructive example of an application of LDRF-PF, we show that no execution of the following program in the $\{X, Y\}$-PF-machine is $\{X, Y\}$-racy, and so $[\![prog]\!]_{\text{PS2.1}} = [\![prog]\!]_{\text{PF}}^{\{X,Y\}}$.

$$\begin{array}{r||l||l}
X^{\texttt{rlx}} := 1 & a := Y^{\texttt{rlx}} & b := Z^{\texttt{rlx}} \\
Y^{\texttt{rel}} := 1 & \textbf{if } a = 1 \textbf{ then} & \textbf{if } b = 1 \textbf{ then} \\
 & \quad Z^{\texttt{rlx}} := 1 & \quad c := X^{\texttt{rlx}}
\end{array} \qquad \text{(MP2)}$$

Clearly, there is no race on Y since the program has no relaxed writes to Y (syntactically). Now, assuming no promises on X and Y, the write to Z in π_2 can neither be promised nor executed before π_1 executes the write to Y. Similarly, the read from X in π_3 cannot be executed before π_2 promises or executes the write to Z. Therefore, the write to Y in π_1 should be first executed in order for π_3 to execute the read from X, and thus there is no X-racy execution in the $\{X, Y\}$-PF-machine.

Proof sketch of LDRF-PF We highlight the main ideas in the proof of Thm. 1, which is the most challenging among our results. For its proof, we introduce an intermediate semantics, called $\mathcal{L}$-*PRF-machine*, and define the notion of race in this machine (PRF stands for promise-read-free).

Definition 6. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the $\mathcal{L}$-*PRF-machine* is the strengthening of PS2.1 obtained by forbidding steps reading from promises to locations in $\mathcal{L}$. We denote by $[\![prog]\!]_{\text{PRF}}^{\mathcal{L}}$ the set of all behaviors of a program *prog* allowed by the $\mathcal{L}$-PRF-machine. $\mathcal{L}$-*racy* executions in the $\mathcal{L}$-PRF-machine are defined exactly as in Def. 5.

Then, we prove the following three lemmas for every program *prog*, from which Thm. 1 directly follows:

(i) $[\![prog]\!]_{\text{PRF}}^{\mathcal{L}} \subseteq [\![prog]\!]_{\text{PF}}^{\mathcal{L}}$.

(ii) If there is no $\mathcal{L}$-racy execution of *prog* in the $\mathcal{L}$-PRF-machine, then $[\![prog]\!]_{\text{PS2.1}} \subseteq [\![prog]\!]_{\text{PRF}}^{\mathcal{L}}$.

(iii) If there is an $\mathcal{L}$-racy execution of *prog* in the $\mathcal{L}$-PRF-machine, then there is one in the $\mathcal{L}$-PF-machine.

Next, we only discuss (ii), and identify an essential property of PS2.1, which we call *promise monotonicity*, that is needed in our proof.

To prove (ii), we use the following "reshuffling" mechanism: when thread π_1 can take a sequence *seq* of thread steps reading a promise m of another thread π_2 to a location $L \in \mathcal{L}$, we first execute π_2 following its certification until it fulfills the promise m and then execute π_1 following *seq* until it reads m. What makes this possible is Lemma 2 below. Using "reshuffling", (ii) is established as follows. Roughly speaking, ignoring the consistency requirement, for the first time a thread can read from a promise on a location in $\mathcal{L}$, we apply the above construction to get an $\mathcal{L}$-racy execution without reading any promise on $\mathcal{L}$ (*i.e.*, a $\mathcal{L}$-racy execution in the $\mathcal{L}$-PRF-machine), which contradicts the premise of (ii). (To meet the consistency requirement, the proof requires repeated applications of the reshuffling.)

Lemma 2 (Promise Monotonicity)**.** Let $\langle \mathcal{T}, M\rangle$ be a (consistent) machine state with a promise m written by thread π_1. Suppose that $\langle \mathcal{T}(\pi_2), M\rangle \to^{*} \xrightarrow{l} \langle T_2, \_\rangle$ for some thread $\pi_2 \neq \pi_1$, label l, and thread state T_2. Then, there exist $l_m \in \{\mathtt{W}(\mathtt{rlx}, m), \mathtt{RMW}(\_, \mathtt{rlx}, \_, m)\}$ and memory M_1 such that:

- $\langle \mathcal{T}, M\rangle \xRightarrow{\pi_1,\_}^{*} \xRightarrow{\pi_1,l_m} \langle \mathcal{T}[\pi_1 \mapsto \_], M_1\rangle$; and
- $\langle \mathcal{T}(\pi_2), M_1\rangle \to^{*} \xrightarrow{l} \langle T_2, \_\rangle$.

Remark 3. Promise consistency does not hold for PS and PS2 since RMW-store reordering breaks it. For *global* DRF-PF, a weaker property, which does hold for PS and PS2, suffices. Specifically, the above reshuffling may break during the execution of π_1 following the sequence *seq* (before it reads m) only if π_1 performs a racy RMW. Global DRF-PF follows from the race on the RMW, but not LDRF-PF since the location of the RMW may not be in $\mathcal{L}$.

2.3.2 Local DRF-RA

To formulate LDRF-RA, we again start by defining a strengthening of PS2.1 w.r.t. a given set of locations.

Definition 7. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-RA-machine* is the strengthening of the $\mathcal{L}$-PF-machine obtained by interpreting all accesses to $\mathcal{L}$ as if they have release/acquire access modes (in (READ-HELPER) and (WRITE-HELPER)). We denote by $[\![prog]\!]^{\mathcal{L}}_{\mathrm{RA}}$ the set of all behaviors of a program *prog* that are allowed by the $\mathcal{L}$-RA-machine.

Next, for stating the premise of LDRF-RA, we introduce the "RA-race-detecting-machine". For that we adopt a "happens-before-based" notion of race, where a necessary condition on the happens-before relation is expressed using the views of the promising semantics. Roughly speaking, the RA-race-detecting-machine invokes UB whenever the machine reaches a state where (i) some thread π is about to read from a location $L \in \mathcal{L}$; (ii) there exists a message m in memory written by some write to L that does not "happen-before" the read (which corresponds to the fact that the view of π for L is strictly lower than the timestamp of m); and (iii) at least one of the write or the read is not annotated as a release/acquire access in the program. This is formalized as follows.

Definition 8. The *$\mathcal{L}$-RA-race-detecting-machine* is the machine obtained from the *$\mathcal{L}$-RA-machine* by adding following thread configuration step:

$$\frac{\begin{array}{c} L \in \mathcal{L} \qquad \lambda \in \{\mathtt{R}(o_{\mathtt{R}}, L, \_), \mathtt{RMW}(o_{\mathtt{R}}, \_, L, \_, \_)\} \qquad \sigma \xrightarrow{\lambda} \_ \\ V(L) < t \qquad m = \langle L @ \_, \_, t \rangle_{\_} \in M \\ o_{\mathtt{R}} = \mathtt{rlx} \lor m \text{ was written by a non-release write}^{9} \end{array}}{\langle\langle \sigma, V, \_\rangle, M\rangle \xrightarrow{\mathsf{race}} \langle\langle \bot, V, \emptyset\rangle, M\rangle}$$

Remark 4. A similar view-based definition of a race can be also used in LDRF-PF. However, such definition would unnecessarily deem too many programs as racy, resulting in a weaker guarantee. For example, with a view-based definition of a race in LDRF-PF, we would not be able to show the absence of $\{X, Y\}$-PF-racy executions for the program in Example 8 (since there is no synchronization from the write to X in π_1 to the read from X in π_3).

[9] Formally, this requires to record the writing access mode in messages.

LDRF-RA is formulated as follows.

Theorem 3 (LDRF-RA)**.** If the `race` transition is never enabled in runs of the $\mathcal{L}$-RA-race-detecting-machine on *prog*, then $[\![prog]\!]_{\text{PS2.1}} = [\![prog]\!]^{\mathcal{L}}_{\text{RA}}$.

Remark 5. When $\mathcal{L} = \texttt{Loc}$, since the $\mathcal{L}$-RA-machine cannot make *any* promise, the race detecting step can be revised as follows (where $\rightarrow$ is the thread step of the $\mathcal{L}$-RA-machine):

$$\frac{\begin{array}{cc}\langle\langle\sigma, V, P\rangle, M\rangle \xrightarrow{l} \langle\langle\sigma', V', P'\rangle, M'\rangle & l \in \{\texttt{R}(o_{\texttt{R}}, \langle L@\_, \_, \_\rangle\_), \texttt{RMW}(o_{\texttt{R}}, \_, \langle L@\_, \_, \_\rangle\_, \_)\} \\ V(L) < t & m = \langle L@\_, \_, t\rangle\_ \in M \\ \multicolumn{2}{c}{o_{\texttt{R}} = \texttt{rlx} \vee m \text{ was written by a non-release write}}\end{array}}{\langle\langle\sigma, V, P\rangle, M\rangle \xrightarrow{\texttt{race}} \langle\langle\sigma', V', P'\rangle, M'\rangle}$$

Then, the global DRF-RA guarantee follows from the local one. The "naive" LDRF-RA discussed in §2.2.2 (which cannot not hold together with all optimizations allowed in PS2.1) formally means to use the above step for race detection in the $\mathcal{L}$-RA-race-detecting-machine.

Example 9. The following example is a variant of the common "load-buffering" test. We show that, using LDRF-RA, this program never exhibits the $a = 1$ outcome.

$$\begin{array}{l|l} a := X^{\texttt{rlx}} \quad /\!/ \neq 1 & b := Y^{\texttt{rlx}} \\ Y^{\texttt{rel}} := 1 & \textbf{if } b = 1 \textbf{ then} \\ & \quad X^{\texttt{rlx}} := 1 \end{array} \qquad \text{(LB-COND)}$$

Assuming RA semantics for X, both the writes to X and to Y cannot be promised, and clearly $a = 1$ is not allowed. Now, we show that the `race` transition is never enabled in executions of this program in the $\{X\}$-RA-race-detecting-machine. Indeed, since the write to X in π_2 can only be executed after the write to Y in π_1 is executed (which cannot be promised because it is a release write), there cannot be any message to X except for the initial message before the read from X in π_1 is executed.

We note that our race condition is strictly stronger (identifying fewer programs as racy) than the standard "happens-before"-based race notion. The latter would deem this program as $\{X\}$-racy. as there is no "happens-before" relation between the accesses to X (since the read of Y is relaxed).

Example 10. We apply LDRF-RA on a location with a write-write race. In the following program, the first two threads access X and Y and raise flags Z and W.

The third thread waits on both flags and then accesses X and Y.

$$\begin{array}{l||l||l} a := X^{\texttt{rlx}} & b := X^{\texttt{rlx}} & \textbf{while}(Z^{\texttt{acq}} + W^{\texttt{acq}} < 2)\ \textbf{do} \\ Y^{\texttt{rlx}} := a + 2 & Y^{\texttt{rlx}} := b + 4 & \quad \textbf{skip} \\ Z^{\texttt{rel}} := 1 & W^{\texttt{rel}} := 1 & X^{\texttt{rlx}} := 1 \\ & & c := Y^{\texttt{rlx}} \quad /\!/\ \mathit{2\ or\ 4} \end{array}$$

While there is a write-write race on Y, there is no *write-read* race on X and Y, and so the `race` transition is never enabled in executions of this program in the $\{X, Y\}$-RA-race-detecting-machine. LDRF-RA ensures that it is safe to assume RA semantics for X and Y. Then, knowing only RA semantics, it follows that $c \in \{2, 4\}$ holds when this program terminates.

Remark 6. To simplify the presentation, we did not discuss release/acquire fences. These allow fine-grained control on the required synchronization, which can improve performance, but results in more races involving relaxed accesses. For the purpose of reasoning about fences using LDRF, we observe that the following transformations do not affect the possible behaviors in the promising semantics:

$$\begin{array}{l} r_1 := X^{\texttt{rlx}} \\ \vdots \\ r_n := X^{\texttt{rlx}} \\ \textbf{fence}^{\texttt{acq}} \end{array} \rightsquigarrow \begin{array}{l} r_1 := X^{\texttt{acq}} \\ \vdots \\ r_n := X^{\texttt{acq}} \\ \textbf{fence}^{\texttt{acq}} \end{array} \qquad \begin{array}{l} \textbf{fence}^{\texttt{rel}} \\ X_0^{\texttt{rlx}} := r_0 \\ X_1^{\texttt{rlx}} := r_1 \\ \vdots \\ X_n^{\texttt{rlx}} := r_n \end{array} \rightsquigarrow \begin{array}{l} \textbf{fence}^{\texttt{rel}} \\ X_0^{\texttt{rel}} := r_0 \\ X_1^{\texttt{srlx}} := r_1 \\ \vdots \\ X_n^{\texttt{srlx}} := r_n \end{array}$$

Programmers may safely use the (better performant) left-hand sides in programs, while assuming the (stronger) semantics provided by the right-hand sides (also for establishing the premise of the LDRF theorem).

2.3.3 Local DRF-SC

The final LDRF guarantee, LDRF-SC, provides the strongest semantics for non-racy accesses, but also requires much more for accesses to be considered non-racy. We note that, unlike C/C++11 [27, 28, 13], the promising semantics does not provide sequentially consistent accesses (it only has SC fences). Thus, a *global* DRF-SC can only pointlessly ensure SC semantics for programs that have no races whatsoever (with no mechanism to actually avoid races). Nevertheless, *local* DRF-SC is still meaningful as it only requires to avoid races on certain locations.

As before, we first define the stronger semantics and the notion of a race.

Definition 9. In the context of a machine state, we call a message *maximal* if there does not exist a message with the same location and higher timestamp. For $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-SC-machine* is the strengthening of the *$\mathcal{L}$-RA-machine* obtained by requiring that for every $L \in \mathcal{L}$:

- reads from L read from maximal messages; and
- writes to L write maximal messages.

We denote by $[\![prog]\!]^{\mathcal{L}}_{\mathrm{SC}}$ the set of all behaviors of a program *prog* that are allowed by the $\mathcal{L}$-SC-machine.

To state the premise of LDRF-SC, we introduce the "SC-race-detecting-machine". It is defined as the RA-race-detecting-machine, except that races may also occur (i) between two RA accesses, and (ii) between two writes.

Definition 10. The *$\mathcal{L}$-SC-race-detecting-machine* is the machine obtained from the $\mathcal{L}$-SC-machine by adding following thread step:

$$\frac{L \in \mathcal{L} \qquad \lambda \in \{\mathtt{R}(\_, L, \_), \mathtt{W}(\_, L, \_), \mathtt{RMW}(\_, \_, L, \_, \_)\} \qquad \sigma \xrightarrow{\lambda} \_ \\ V(L) < t \qquad m = \langle L@\_, \_, t\rangle_{\_} \in M}{\langle\langle \sigma, V, \_\rangle, M\rangle \xrightarrow{\mathtt{race}} \langle\langle \bot, V, \emptyset\rangle, M\rangle}$$

Then, LDRF-SC is formulated as follows.

Theorem 4 (LDRF-SC)**.** If the `race` transition is never enabled in runs of the $\mathcal{L}$-SC-race-detecting-machine on *prog*, then $[\![prog]\!]_{\mathrm{PS2.1}} = [\![prog]\!]^{\mathcal{L}}_{\mathrm{SC}}$.

Example 11. Consider the message passing program:

$$\begin{array}{l} D^{\mathtt{rlx}} := 42 \\ F^{\mathtt{rel}} := 1 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := F^{\mathtt{acq}} \\ \textbf{if } a = 1 \textbf{ then} \\ \quad a := D^{\mathtt{rlx}} \quad /\!/ 42 \end{array} \qquad \text{(MP)}$$

In all its executions in the $\{D\}$-SC-race-detecting-machine, the view of π_2 for D after reading 1 from F points to the message $D = 42$ written by π_1. Therefore, the `race` transition is never enabled for this program in the $\{D\}$-SC-race-detecting-machine. Then, LDRF-SC with $\mathcal{L} = \{D\}$ ensures SC semantics on the location D.

2.3.4 Time-wise Local DRF Guarantees

In this subsection, we demonstrate generalized LDRF guarantees, namely "time-wise LDRF guarantees." Specifically, we provide two theorems, time-wise LDRF-PF (LDRF-PF-Time) and time-wise LDRF-SC (LDRF-SC-Time). Roughly speaking, time-wise LDRF guarantees say that, given a machine state Σ, the next possible machine steps in PS are equivalent to those in certain stronger semantics provided that Σ is not a racy state in the stronger semantics. Since these guarantees are applied to a machine state instead of the full execution of a program, they do not require race-freedom of the machine steps taken before reaching the given machine state. Moreover, starting from a given machine state, one can assume a stronger semantics until it reaches a racy machine state.

It is important to note that unlike Thm. 4 where the SC semantics denotes a strengthening of RA accessing only the latest messages to each location, the SC semantics of LDRF-SC-Time is not RA-synchronizing, *i.e.*, it is a strengthening of PF accessing only the latest messages to each location. Indeed, this is very similar to the local SC semantics of LDRF of [18]. Though we have another formulation of time-wise LDRF-SC with the SC semantics accompanying RA synchronization, which we believe is provable, we do not present it as the theorem is too complicated due to the presence of RA synchronizations. For the same reason, we do not present the "time-wise LDRF-RA" theorem as well.

To formulate LDRF-PF-Time, we first define an $\mathcal{L}$-racy machine state. Roughly, a machine state Σ is $\mathcal{L}$-racy if (i) starting from Σ, a thread can take multiple steps and write a message m to a location in $\mathcal{L}$ by a relaxed write; and (ii) immediately after m is written, another thread can take multiple steps and read the message m.

Definition 11. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, a machine state $\Sigma = \langle \mathcal{T}, M \rangle$ is *$\mathcal{L}$-racy* if there exist a location $X \in \mathcal{L}$, two different threads $\pi_1 \neq \pi_2$, a memory M', labels l_1, l_2, and a message m such that:

- $\langle \mathcal{T}, M\rangle \xRightarrow{\pi_1,\_}{}^{*} \xRightarrow{\pi_1,l_1} \langle \_, M_1\rangle$
- $\langle \mathcal{T}(\pi_2), M_1\rangle \rightarrow^{*} \xrightarrow{l_2} \langle \_, \_\rangle$
- $l_1 \in \{\texttt{W}(\texttt{rlx}, m), \texttt{RMW}(\_, \texttt{rlx}, \_, m)\} \wedge l_2 \in \{\texttt{R}(\_, m), \texttt{RMW}(\_, \_, m, \_)\}$

Using the above definition, LDRF-PF-Time is formulated as follows. The main difference of LDRF-PF-Time from LDRF-PF is that it does not consider racy steps taken before reaching Σ, to which LDRF-PF-Time applies. Moreover, one can stop applying LDRF-PF-Time whenever the machine reaches a racy state.

Theorem 5 (LDRF-PF-Time)**.** Given a set $\mathcal{L} \subseteq \mathsf{Loc}$ and a machine state $\langle \mathcal{T}, M\rangle$ which has no promise to locations in $\mathcal{L}$, if $\langle \mathcal{T}, M\rangle$ has some execution in the PS2.1 with a final outcome R, then one of the following holds.

- $\langle \mathcal{T}, M\rangle$ has some execution in the $\mathcal{L}$-PF-machine with the outcome R.
- $\langle \mathcal{T}, M\rangle$ can reach a $\mathcal{L}$-racy state $\langle \mathcal{T}', M'\rangle$ in the $\mathcal{L}$-PF-machine, and $\langle \mathcal{T}', M'\rangle$ has some execution in the PS2.1 with the outcome R.

As we previously described, the SC semantics of LDRF-SC-Time varies from that of Thm. 4. Therefore, we first define the SC machine and the SC-race-detecting-machine of LDRF-SC-Time.

Definition 12. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-SC-machine* is the strengthening of the *$\mathcal{L}$-PF-machine* obtained by requiring that for every $L \in \mathcal{L}$:

- reads from L read from maximal messages; and
- writes to L write maximal messages.

Definition 13. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-SC-race-detecting-machine* is the machine obtained from the $\mathcal{L}$-SC-machine by adding following thread step invoking UB:

$$\frac{\begin{array}{c} L \in \mathcal{L} \qquad l \in \{\texttt{R}(\_, L, \_), \texttt{W}(\_, L, \_), \texttt{RMW}(\_, \_, L, \_, \_)\} \\ \sigma \xrightarrow{l} \_ \qquad m = \langle L@\_, \_, t\rangle_{\_} \in M \qquad V(L) < t \end{array}}{\langle\langle \sigma, V, \_\rangle, M\rangle \xrightarrow{\texttt{race}(m)} \langle\langle \bot, V, \emptyset\rangle, M\rangle}$$

Then, we define a racy machine state. Note that our definition of a racy machine state is weaker than that of [18] (*i.e.*, our definition identifies more machine state to be racy than that of [18]) due to the presence of promises.

Definition 14. Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, a machine state $\Sigma = \langle \mathcal{T}, M \rangle$ is *$\mathcal{L}$-racy* if one of the following holds.

(i) There exist a location $X \in \mathcal{L}$ and two different threads $\pi_1 \neq \pi_2$, thread state $\mathcal{T}'$, memory M' and a label l, and message m such that:

- $\langle \mathcal{T}, M \rangle \overset{\pi_1,\_}{\Longrightarrow}{}^{*} \overset{\pi_1,l}{\Longrightarrow} \langle \mathcal{T}', M_1 \rangle$
- $l \in \{\mathtt{W}(\mathtt{rlx}, m), \mathtt{RMW}(\_, \mathtt{rlx}, \_, m)\}$
- $\langle \mathcal{T}', M_1 \rangle$ can take machine steps $\langle \mathcal{T}', M_1 \rangle \overset{\pi_2,\_}{\Longrightarrow}{}^{*} \xRightarrow{\pi_2,\mathtt{race}(m)} \_$ in the $\mathcal{L}$-SC-race-detecting-machine.

(ii) There exist a thread π and message m such that:

- $\langle \mathcal{T}, M \rangle$ can take machine steps $\langle \mathcal{T}, M \rangle \overset{\pi,\_}{\Longrightarrow}{}^{*} \xRightarrow{\pi,\mathtt{race}(m)} \_$ in the $\mathcal{L}$-SC-race-detecting-machine.

Finally, LDRF-SC-Time is formulated as follows.

Theorem 6 (LDRF-SC-Time)**.** Given a set $\mathcal{L} \subseteq \mathsf{Loc}$ and a machine state $\langle \mathcal{T}, M \rangle$ which has no promise on locations in $\mathcal{L}$, if $\langle \mathcal{T}, M \rangle$ has some execution in the PS2.1 with a final outcome R, then one of the following holds.

- $\langle \mathcal{T}, M \rangle$ has some execution in the $\mathcal{L}$-SC-machine with the outcome R.
- $\langle \mathcal{T}, M \rangle$ can reach a $\mathcal{L}$-racy state $\langle \mathcal{T}', M' \rangle$ in the $\mathcal{L}$-SC-machine, and $\langle \mathcal{T}', M' \rangle$ has some execution in the PS2.1 with the outcome R.

2.4 Applying LDRF for Modular Reasoning

In this section, we outline several applications of the local DRF guarantees (focusing on LDRF-RA) for client and library developer reasoning. Roughly speaking, local DRF is essential for modular reasoning because it ensures the absence of certain behaviors in which unrelated pieces of code affect one another. Without a local DRF guarantee, it might be that some completely orthogonal calls to library code (such as the call to a logging function in the second example below) allow additional behaviors of the client's code! We believe that ignoring unrelated races in library calls (*e.g.*, in the implementations of synchronization mechanisms or in debugging code) is widely informally done in practice, and view the local DRF guarantees as providing formal justifications for this kind of intuitive reasoning.

We start by observing that the $\mathcal{L}$-PF-machine and the $\mathcal{L}$-X-race-detecting-machines for $X \in \{\mathrm{RA}, \mathrm{SC}\}$ all enjoy a useful *locality property* making it safe to completely ignore code not accessing locations in $\mathcal{L}$ when reasoning about code only accessing locations in $\mathcal{L}$. Indeed, since promises to $\mathcal{L}$ are banned in those machines, a step that executes code not accessing locations in $\mathcal{L}$ can only increase the thread view on locations in $\mathcal{L}$, or add reservations for locations in $\mathcal{L}$. These two effects only decrease the possible behaviors (including the ability to detect a race), so it is safe to ignore them when reasoning about code only accessing $\mathcal{L}$. (For this reason, clients using the LDRF results do not need to understand the notion of reservation.)

2.4.1 Reasoning About Client Code

We show typical cases of client RA-centric reasoning using LDRF-RA.

Synchronization with Lock Consider the following program that uses a lock and a collection libraries.

$$\begin{array}{l||l||l} & r_0 := \textbf{pop\_wait}() & r_1 := \textbf{pop\_wait}() \\ \textbf{push}(5) & \textbf{lock}() & \textbf{lock}() \\ \textbf{push}(7) & s_0 := S^{\mathtt{rlx}} & s_1 := S^{\mathtt{rlx}} \\ & S^{\mathtt{rlx}} := s_0 + r_0 & S^{\mathtt{rlx}} := s_1 + r_1 \\ & \textbf{unlock}() & \textbf{unlock}() \end{array}$$

Suppose that **lock**() and **unlock**() are specified by the following RA specification (the implementation may be more efficient, but the library guarantees that it behaves the same as the following specification in PS2.1):

```
lock() ≜                                  unlock() ≜
  do $a :=$ CAS$^{\texttt{acqrel}}(L, 0, 1)$      $L^{\texttt{rel}} := 0$
  while $(a \neq 0)$
```

Further, suppose that the collection library guarantees that **push** and **pop_wait** (syntactically) never access S and L, and that when the same number of push and pop are invoked, the values returned by **pop_wait** are in some one-to-one correspondence with the values pushed by **push**.[10]

To use LDRF-RA the client has to show that this program has no racy execution in the $\{S, L\}$-RA-race-detecting-machine. The reasoning is straightforward, and can be done only knowing the RA semantics:

(i) By the locality property, we can safely ignore the impact on S and L by **push** and **pop_wait**;

(ii) Since L is only accessed by RA accesses, we know that there are not any races on L;

(iii) The lock specification (specifically the RA synchronization from **unlock**() to **lock**()) ensures that a thread accessing S always has the maximal view on S, so the accesses to S are not racy as well.

Then, by LDRF-RA, the client may safely assume the $\{S, L\}$-RA-machine. Hence, again by the locality property and using the collection specification, it easily follows that the final value of S is 12 $(= 5 + 7)$.

We note that the above standard reasoning is only justified by LDRF-RA. Since the collection library may not have an RA-based specification (unlike the lock library), global DRF-RA cannot be applied to reach the above conclusion.

[10]The library may assume that the client code does not invoke UB, which is the case in our example.

Synchronization with Queue Next, we consider an example that uses a queue and a log libraries. For an array U of size 32×64, the first thread repeats the following for $0 \le i \le 31$: write some data to $U[i \times 64, \dots, i \times 64 + 63]$ via **write**(U, i), put the index i in the queue via **enque**(i), and log the result via **log**(s). Each other thread takes an index from the queue via **try_deque**$()$, logs the result via **log**(i), and if successful, uses the data in $U[i \times 64, \dots, i \times 64 + 63]$ via **use**(U, i) that only reads from (and possibly writes to) $U[i \times 64, \dots, i \times 64 + 63]$. Here **log** is an unspecified racy library function that accesses a disjoint set of locations.

```
for $i$ in (0 to 31)    ||  $i$ = try_deque()     ||       ||  $i$ = try_deque()
    write($U, i$)       ||  log($i$)              ||  ...  ||  log($i$)
    $s$ := enque($i$)   ||  if $i \ge 0$ then     ||       ||  if $i \ge 0$ then
    log($s$)            ||      use($U, i$)       ||       ||      use($U, i$)
```

Suppose that **enque** and **try_deque** are specified by the following RA specification (ignore the parentheses around some `acq` and `rel` for now).

```
enque($d$) $\triangleq$                       try_deque() $\triangleq$
    lock()                                    $t := T^{\texttt{acq}}$
    $t := T^{(\texttt{acq})}$                 $b := B^{\texttt{acq}}$
    if not $t < 32$ then                      if not $b < t$ then
        unlock(); return full                     return empty
    $D[t]^{(\texttt{rel})} := d$              $d := D[b]^{(\texttt{acq})}$
    $T^{\texttt{rel}} := t + 1$               $b' := \mathbf{CAS}^{\texttt{acqrel}}(B, b, b+1)$
    unlock(); return 0                        return $(b = b'\ ?\ d\ :\ \texttt{fail})$
```

The queue library uses a static (non-circular) buffer D of size 32 and two locations T and B (initialized to 0) that point to the top and bottom indices of the queue, where **enque**(d) puts the data d to the top and **try_deque**$()$ takes a data from the bottom. While **try_deque** is non-blocking, **enque** uses the lock specified above to avoid race between **enque**'s.

To use LDRF-RA the client has to show that this program has no racy execution in the $\{U, D, T, B, L\}$-RA-race-detecting-machine. The reasoning is as follows only knowing the RA semantics:

(i) By the locality property, we can safely ignore the impact on U, D, T, B, L by **log**;

(ii) Since D, T, B, L are only accessed by RA accesses, there are not any races on them;

(iii) The queue specification ensures a synchronization from an **enque** writing to $D[k]$ to a **try_deque** reading from $D[k]$ for any k via the accesses to T, since the **enque** writes $k+1$ to T, the **try_deque** reads some $k' > k$ from T, and all the writes to T are synchronized via **lock**() and **unlock**();

(iv) It also ensures that each successful **try_deque** returns a unique index due to the atomicity of **CAS** in **try_deque**;

(v) From these, it follows that each **use**(U, i) accesses disjoint locations, and since the synchronization on T ensures no races on U between write **write**(U, i) and **use**(U, i), we avoid races on U as well.

Then, by LDRF-RA, the client may safely assume the semantics provided by the $\{U, D, T, B, L\}$-RA-machine. We again note that due to the presence of **log**, global DRF-RA cannot be applied here.

2.4.2 Reasoning About Library Code

Next, we describe how LDRF-RA can be used to reason about the implementation of the queue library above. We consider an implementation of the above specification that simply lowers the accesses in parentheses, (`acq`) and (`rel`), to be `rlx` accesses. (This optimization may be significant if the size of each cell in D is large.)

By applying LDRF-RA for $\{D, T, B, L\}$, one shows that the implementation meets the specification under *an arbitrary context* that does not access D, T, B, L (again knowing nothing beyond RA):

(i) By the locality property, we can safely ignore the impact on D, T, B, L by the context;

(ii) Since B, L are only accessed by RA accesses, there are not any races on them;

(iii) For T, the only possible race is between the `rlx` read and the `rel` write in **enque**, which, however, reside in the same locked region thereby avoiding race;

(iv) For D, the reasoning in §2.4.1(iii) for the client program applies, thereby avoiding races on $D[k]$ for any k.

Then, by LDRF-RA, the library developer may safely assume the $\{D, T, B, L\}$-RA-machine, whose behaviors are included in those of the queue specification, and thus we can complete the verification. Note that since the context can be racy, global DRF-RA cannot be applied here.

We note that by using LDRF-PF, it is possible to slightly improve the above implementation, in the price of reasoning in the PF-machine instead of the RA-machine. Indeed, the read from B in **try_deque**() can be made relaxed, and the **CAS** on B can be made `rel` (or `srlx`) because LDRF-PF does not require any condition on reads. Then, for any program *prog* that uses this implementation, a similar argument shows that there are no $\{D, T, B, L\}$-racy executions in the $\{D, T, B, L\}$-PF-machine, and it follows that $[\![prog]\!]_{\text{PS2.1}} = [\![prog]\!]^{\{D,T,B,L\}}_{\text{PF}}$.

2.5 Mapping PS2.1 to Hardware

In this section we discuss the mapping from PS2.1 to mainstream architectures, and evaluate the performance impact of forbidding RMW-store reordering.

PS2.1 supports the intended compilation schemes to mainstream architectures [29], but for Armv8, it requires an additional (fake) control dependency from the read part ("load-linked") of each fetch-and-add and exchange instruction with relaxed read mode.

The compilation schemes for these instructions along with the more optimal schemes are given in Fig. 2.4. The code in red indicates manually inserted fake control dependencies. Note that the compiled Armv8 assembly assumes that the address of

RMW operation	Armv8 assembly		
	Optimal (LLVM 10.0.0)	PS2.1 (Scheme 1)	PS2.1 (Scheme 2)
$c := \mathbf{FADD}(X, a)$	`.L:`	`.L:`	`.L:`
	`ldxr x1, [x2]`	`ldxr x1, [x2]`	`ldxr x1, [x2]`
	`add x1, x1, x3`	`add x1, x1, x3`	`add x1, x1, x3`
	`stxr w1, x1, [x2]`	`stxr w1, x1, [x2]`	`stxr w1, x1, [x2]`
	`cbnz w1, .L`	`cbnz w1, .L`	`cbnz w1, .L`
		`cbnz x1, .LFAKE`	`cmp x1, x1`
		`.LFAKE:`	`beq .LFAKE`
			`.LFAKE:`
$c := \mathbf{XCHG}(X, a)$	`.L:`	`.L:`	`.L:`
	`ldxr x1, [x2]`	`ldxr x1, [x2]`	`ldxr x1, [x2]`
	`stxr w1, x3, [x2]`	`stxr w1, x3, [x2]`	`stxr w1, x3, [x2]`
	`cbnz w1, .L`	`cbnz w1, .L`	`cbnz w1, .L`
		`cbnz x1, .LFAKE`	`cmp x1, x1`
		`.LFAKE:`	`beq .LFAKE`
			`.LFAKE:`

Figure 2.4: Compilation schemes of RMW opearations to Armv8 architecture

the memory location X is stored in a register `x2` and the value of a is stored in a register `x3`.

[11] Lee *et al.* [4, Section 6.5] established (in Coq) the correctness of these schemes from PS2 to hardware models using the Intermediate Memory Model, IMM [31]. We observe here that their proof works as is for PS2.1.[12] We note that compared to PS, PS2.1 still supports more efficient mapping of RMW operations, which for PS require an "ld fence" barrier that is more expensive than a control dependency (see Example 2).

We believe that forbidding RMW-store reorderings only mildly affects performance since: (i) standard compilers do not aggressively reorder RMWs with atomic writes [21]; (ii) with the exception of Armv8, mainstream hardware (x86-TSO, POWER, Armv7, and RISC-V) do not reorder such accesses; and (iii) the performance overhead in Armv8 for forbidding this optimization is negligible.

We demonstrate (iii) by evaluating 19 highly concurrent data structures with

[11]Our compilation schemes employ standard LL/SC-style RMW implementations. We leave to future work the evaluation of an implementation that uses Armv8.1's LSE (Large System Extension) for RMWs [30].

[12]The proof does not handle atomic exchange instructions, which are not supported in IMM.

Benchmark	Scheme	Average (%)
libcds, 32 threads	(A)	-0.15 (± 5.44)
	(B)	0.10 (± 5.53)
libcds, 128 threads	(A)	0.03 (± 4.48)
	(B)	0.10 (± 4.51)
FADD litmus test	(A)	0.32 (± 2.81)
	(B)	1.05 (± 2.87)
FADD-RW litmus test	(A)	-0.09 (± 3.56)
	(B)	0.22 (± 3.51)
XCHG litmus test	(A)	-0.71 (± 2.73)
	(B)	-0.00 (± 2.74)
XCHG-RW litmus test	(A)	1.09 (± 3.97)
	(B)	0.22 (± 3.79)

Figure 2.5: Performance overhead for each benchmark (The "Average" column denotes the arithmetic mean and the 95% confidence interval.)

extensive use of fetch-and-add and exchange operations selected from the CDS C++ library [32], as well as four artificial "worst case litmus tests" that repeatedly perform fetch-and-add and exchange operations. Specifically, the four tests consist of following: FADD/XCHG tests where each thread repeatedly performs **FADD/XCHG** to a single location; and FADD-RW/XCHG-RW tests where each thread repeatedly performs **FADD/XCHG** to a single location followed by load/store to another location ($n/2$ threads load and $n/2$ threads store).

The benchmarks are compiled with LLVM 10.0.0 with manual insertion of fake conditional branches to fetch-and-add and exchange instructions using two different schemes: (A) direct branch on the loaded value; or (B) compare the loaded value with itself and branch on the comparison result. The latter requires an extra `cmp` instruction, but more likely to be optimized by branch speculations as it jumps deterministically. For the evaluation, we used 2 socket, 64-core 2.5GHz ThunderX2 64-bit Armv8 server with 128GB memory. We ran each benchmark 360 times and discarded the 30 fastest and 30 slowest results among them.

Figure 2.5 summarizes the performance overhead for each benchmark. We conclude that there is no statistical evidence for a noticeable performance cost induced by the suboptimal RMW compilation of PS2.1. Detailed results of the performance evaluation in Fig. 2.5 is given in Fig. 2.6. The error bars in each figure represent 95% confidence intervals.

Remark 7. During the evaluation, we identified a bug in LLVM's compilation of exchange instructions to Armv8. When the value read by the exchange instruction is never used, LLVM 10.0.0 compiles C++11 relaxed exchange instructions into Arm's plain *store* instructions. However, since an acquire fence may induce synchronization when it follows an exchange instruction, but *not* when it follows a store, this optimization is unsound: it may introduce behaviors in the compiled Armv8 assembly that are not allowed for the C++ source program.

During the performance evaluation, we identified a bug in LLVM's compilation of exchange instructions to Armv8.

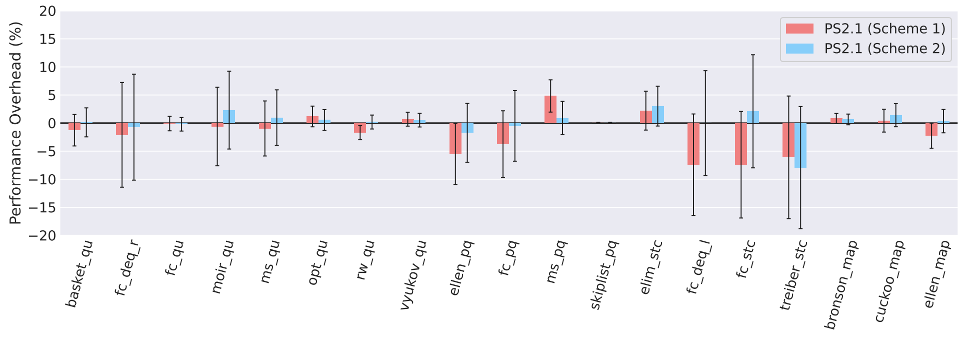
To see a concrete example of the miscompilation, consider the following program and its mapping to Armv8 (for simplicity, we only show the mapping of T_2):

$$\begin{array}{l||l} Y^{\texttt{rlx}} := 1 & \_ := \textbf{XCHG}(X, 42) \\ a := \textbf{FADD}^{\texttt{rel}}(X, 1) \quad /\!\!/\, 0? & \textbf{fence}^{\texttt{acq}} \\ & b := Y^{\texttt{rlx}} \quad /\!\!/\, 0? \end{array} \qquad \rightsquigarrow \qquad \begin{array}{l} \texttt{str x1, [x2]} \\ \texttt{dmb.ishld} \\ \texttt{ldr x0, [x3]} \end{array}$$

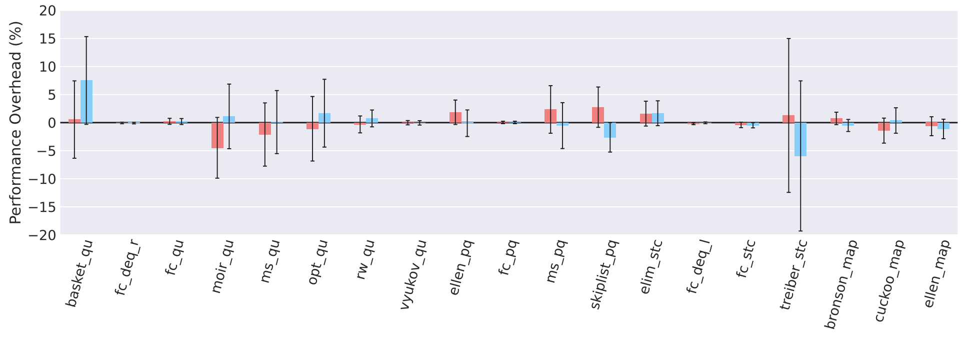
The annotated behavior $a = 0$ and $b = 0$ is not allowed in the source program following the C/C++11 memory model [27]. Once T_1 updates X from the initial value 0 to 1, T_2 can only exchange 1 with 42 due to the atomicity of the fetch-and-add instruction. Then, the acquire fence induces *happens-before* order from $Y = 1$ write by T_1 to the read from Y by tid_2. Therefore, T_2 is only allowed to read 1 from Y, but not from the initial write 0. However, as the optimization entirely removed the effect of the read by the exchange instruction, $a = 0$ and $b = 0$ is allowed in the compiled assembly by Armv8 semantics [6].

2.6 A Counterexample to Local DRF in PS

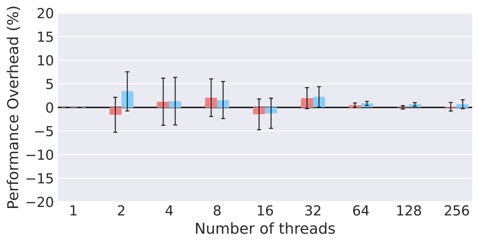
In this section, we demonstrate a counterexample to LDRF-PF, LDRF-RA, and LDRF-SC in PS as well as in PS2. Though we center our discussion around LDRF-PF, it should be clear that a similar narrative works for LDRF-RA and LDRF-SC. As in §2.2.1, we first show a program that is $\mathcal{L}$-PF-race-free for a given set of locations $\mathcal{L}$ while a sequence of optimizations entails an outcome that cannot occur in the



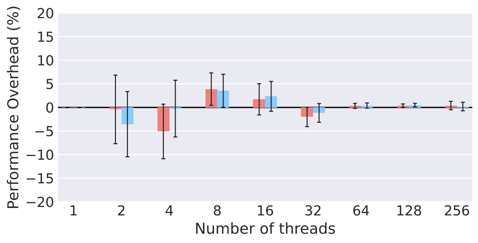
(a) Performance of 19 data structures from libcds run in 32 threads



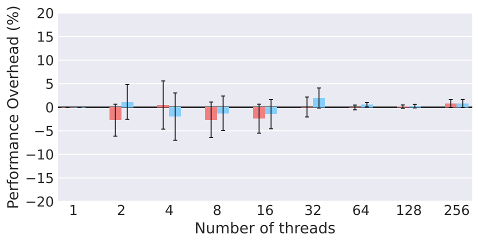
(b) Performance of 19 data structures from libcds run in 128 threads



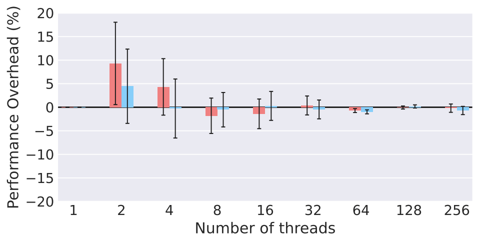
(c) Performance of FADD litmus test



(d) Performance of FADD-RW litmus test



(e) Performance of XCHG litmus test



(f) Performance of XCHG-RW litmus test

Figure 2.6: Performance of benchmarks on 64-bit Armv8 machine

$\mathcal{L}$-PF-machine.

```
$a := X^{\texttt{rlx}}$                      ||  $b := Z^{\texttt{rlx}}$
if $a = 1$ then                              ||  if $b = 0$ then
    $\_ := \textbf{FADD}(W, 1)$              ||      $X^{\texttt{rlx}} := 1$
    $Y^{\texttt{rlx}} := 1$                  ||  else
    $Z^{\texttt{rlx}} := 1$                  ||      $c := \textbf{FADD}(W, 1)$   // 0
                                             ||      if $c = 0$ then
                                             ||          $d := Y^{\texttt{rlx}}$
                                             ||          $X^{\texttt{rlx}} := d$
                                             ||      else
                                             ||          $X^{\texttt{rlx}} := 1$
```
(LDRF-Fail-PS)

Here, we show that there is no $\{Y\}$-PF-racy execution of the LDRF-Fail-PS program in the $\{Y\}$-PF-machine (of PS or PS2.1). Indeed, in any execution of the LDRF-Fail-PS program in $\{Y\}$-PF-machine, $c := \textbf{FADD}(W, 1)$ of π_2 cannot read 0 from W. Thus, tid_2 never reads from Y, and the program is $\{Y\}$-race-free. Specifically, in order for π_2 to enter the else-branch on $b = 0$, π_2 should be able to read a non-zero value from Z. For this, π_2 should first promise $X = 1$, allowing π_1 to either promise or write $Z = 1$. After promising $X = 1$, π_2 requires $Y = 1$ message in the memory in order to fulfill its promise $X = 1$ while entering the if-branch on $c = 0$. Since making a promise to Y is not allowed in the $\{Y\}$-PF-machine, π_1 should execute the **FADD** to W, updating it from 0 to 1, before executing the write 1 to Y. Then, $c := \textbf{FADD}(W, 1)$ of π_2 can only read 1 from W, meaning that $d = Y$ cannot ever be executed. Therefore, LDRF-Fail-PS is $\{Y\}$-PF-race-free and $c := \textbf{FADD}(W, 1)$ of π_2 can only read 1 from W, but not 0.

However, the following sequence of transformations entails an execution where $c := \textbf{FADD}(W, 1)$ reads 0 from W as shown in Fig. 2.7

Therefore, the LDRF guarantees do not hold for PS and PS2. Note that the problematic execution takes a different path to the certification when a promise $(X = 1)$ was made, like the OOTA4 example in [33] and the Coh-CYC example in [8].

The compiler may optimize Thread 1 as shown below:

(1)
```
$a := X^{\mathrm{rlx}}$
if $a = 1$ then
  $Y^{\mathrm{rlx}} := 1$
  $Z^{\mathrm{rlx}} := 1$
  $\_ := \mathbf{FADD}(W, 1)$
```

(2)
```
$e := Z^{\mathrm{rlx}}$
if $e = 0$ then
  $f := Y^{\mathrm{rlx}}$
  if $f = 1$ then
    $b := Z^{\mathrm{rlx}}$
    if $b = 0$ then
      $X^{\mathrm{rlx}} := 1$
    else
      $c := \mathbf{FADD}(W, 1)$
      if $c = 0$ then
        $d := Y^{\mathrm{rlx}}$
        $X^{\mathrm{rlx}} := d$
      else
        $X^{\mathrm{rlx}} := 1$
  else
    $b := Z^{\mathrm{rlx}}$
    if $b = 0$ then
      $X^{\mathrm{rlx}} := 1$
    else
      ...
else
  ...
```

(3)
```
$e := Z^{\mathrm{rlx}}$
if $e = 0$ then
  $f := Y^{\mathrm{rlx}}$
  if $f = 1$ then
    $b := Z^{\mathrm{rlx}}$
    if $b = 0$ then
      $X^{\mathrm{rlx}} := 1$
    else
      $c := \mathbf{FADD}(W, 1)$
      if $c = 0$ then
        (eliminated)
        $X^{\mathrm{rlx}} := 1$
      else
        $X^{\mathrm{rlx}} := 1$
  else
    (eliminated)
    (eliminated)
    $X^{\mathrm{rlx}} := 1$

    (eliminated)
else
  ...
```

(4)
```
$e := Z^{\mathrm{rlx}}$
if $e = 0$ then
  $f := Y^{\mathrm{rlx}}$
  if $f = 1$ then
    $b := Z^{\mathrm{rlx}}$
    if $b = 0$ then
      $X^{\mathrm{rlx}} := 1$
    else
      $X^{\mathrm{rlx}} := 1$
      $c := \mathbf{FADD}(W, 1)$
  else
    $X^{\mathrm{rlx}} := 1$
else
  ...
```

(5)
```
$e := Z^{\mathrm{rlx}}$
if $e = 0$ then
  $X^{\mathrm{rlx}} := 1$
  $f := Y^{\mathrm{rlx}}$
  if $f = 1$ then
    $b := Z^{\mathrm{rlx}}$
    if $b \neq 0$ then
      $c := \mathbf{FADD}(W, 1)$
else
  ...
```

(1) reorder the independent **FADD** on W and writes to Y and Z.

Now, the compiler may optimize Thread 2 as shown above:

(2) introduce reads $e := Z$ and $f := Y$ and insert dummy if-then-else branches on $e = 0$ and $f = 1$;

(3) forward the read $f := Y$ to the read $d := Y$ in the if-branch, turning $X := d$ into $X := 1$, and forward the read $e := Z$ to the read $b := Z$ in the else-branch, turning it into $b := 0$;

(4) merge the branch on $c = 0$ and reorder the write $X := 1$ with $c := \mathbf{FADD}(W, 1)$;

(5) hoist the common writes $X := 1$ in the branch on $b = 0$ and reorder it above $f := Y$.

The program after the transformation exhibits such a behavior in $\{Y\}$-PF-machine as follows:

(i) π_2 promises $X = 1$.

(ii) π_1 reads 1 from X and writes $Y = 1$.

(iii) π_2 reads $Z = 0$, enters the if-branch on $e = 0$, fulfills the promise $X = 1$, reads $Y = 1$, and enters the if-branch on $f = 1$.

(iv) π_1 writes $Z = 1$.

(v) π_2 reads 1 from Z, enters if-branch on $b \neq 0$, and updates W from 0 to 1.

(vi) The both threads execute the rest of their instructions.

Figure 2.7: Program transformations and execution of LDRF-Fail-PS

2.7 Conclusion and Related Work

Studying local DRF guarantees in a fully relaxed semantics, we have achieved a negative and a positive outcomes. The negative one is an unfortunate impossibility result: standard compiler optimizations are inconsistent with local DRF guarantees. On the positive side, local DRF can be achieved by giving up certain RMW-store reorderings, which carries no meaningful performance penalty. The positive result is established constructively by showing a variant of the promising semantics that satisfies the standard optimizations intended to be sound in weak memory models except for RMW-store reorderings, and validates several local DRF guarantees.

We believe that it may be useful to study existing and novel models through the lens of our results also beyond the context of the promising semantics. A "just right" programming language shared-memory concurrency model that is not too strong to allow efficient implementation and not too weak to program with has been the subject of extensive research in recent years (*e.g.*, [34, 35, 27, 20, 36, 37, 8, 38, 39, 13, 40, 22, 41, 18, 42, 23, 33, 43]). While implementability is nowadays relatively well-defined, the criteria for the programmability aspect are much less evident. Since, as we argue in this chapter, local DRF guarantees facilitate modular software development, these guarantees provide better programmability desideratum than the standard global DRF properties. Our impossibility result shows an inherent trade-off that has to be considered when designing a memory model, while the positive result assigns the blame on RMW-store reorderings.

Several papers have previously studied local DRF guarantees. Dolan *et al.* [18] introduced local DRF-SC, and established such guarantees for a model with two types of access. Their guarantees account for two aspects of "locality": (*i*) in terms of "space", which is similar to our location-wise LDRF above and (*ii*) in terms of "time", which we cover in §2.3.4. However, their memory model is much stronger than the one studied here. In particular, it is an "in-order" model (see Def. 3), as even their weak accesses completely forbid load-store reorderings (including RMW-store reorderings), and cannot be compiled to plain machine accesses on architectures like Arm. In addition, their strong accesses are stronger than C11's SC accesses, so that strong stores have to be mapped to atomic exchanges even on x86.

Dongol *et al.* [19] established local DRF-SC guarantees for a model more general than the one of [18] with multiple access modes. In their model, threads synchronize via software transactions with RA semantics. While release/acquire RMWs can be implemented as transactions, the problematic relaxed RMWs are not expressible in the model of [19], so that our impossibility result does not apply.

Recently, Jagadeesan *et al.* [33] presented a denotational concurrency model and sketched (without full proofs) a time-wise local DRF-SC guarantee for a fragment of this model that does not include fences and RMWs. (They presented several variants, and their reported LDRF result is for a version that does not support irrelevant load introduction.) Their model is multi-copy-atomic, and thus, unlike PS2.1, it cannot be efficiently compiled to POWER or Armv7.

We note that the strengthening of accesses in the "SC machines" (used for detecting races for the LDRF-SC premise) in prior work [18, 19, 33] does not make them synchronizing (inducing "happens-before" w.r.t. other locations). Thus, like ours, previous local DRF-SC theorems are weaker than the naive formulation discussed in §2.2.2.

Finally, while modular reasoning about libraries in weak memory semantics has been studied in multiple papers, *e.g.*, [44, 45, 46, 17], to the best of our knowledge, the observation that location-wise local DRF guarantees are essential for such reasoning is lacking in prior work. We leave to future work the development of LDRF-based formal tools, which will allow one to formalize (and possibly mechanize) reasoning as we did in §2.4. In particular, our LDRF-PF paves the way for the application of program logics for an "in-order" semantics (*e.g.*, the logic in [15] that essentially targets the PF model), which is significantly simpler than any semantics allowing load-buffering behaviors. We also note that while the applications in §2.4 are for RA-centric specifications, our local DRF results are generally applicable for weaker library specifications as well. Nevertheless, it is currently unclear how to formally specify and verify libraries in memory models that allow load buffering behaviors (as was studied in [17] for 'in-order' models). We leave this question as well for future work.

Chapter 3

Sequential Reasoning for Compiler Writers

3.1 Introduction: Optimizations under Weak Memory

Weak memory models aim to support a wide range of source-to-source compiler optimizations. These optimizations provide indispensable means for improving performance, especially the optimizations involving memory accesses intended to be non-racy ("non-atomics" in C/C++), which are more frequent and allow more optimizations compared to synchronization accesses ("atomics" in C/C++).

The soundness of compiler optimizations is a contextual refinement property—the transformed piece of code should behave as prescribed by the semantics of its source under any context. For certain optimizations, mostly access reorderings and redundant access eliminations, soundness was established under multiple concrete weak memory models of different kinds [47, 20, 35, 48, 4, 8, 49, 3, 23, 33, 37, 50]. These results, however, require delicate and fragile arguments that depend on the full underlying complex memory model, which is often very different than standard operational semantics.[1] This poses a significant challenge for compiler optimization developers, especially in the context of certified optimizing compilers, notably CompCert [53, 54], whose simulation-based approach for the soundness of each optimization pass cannot accommodate complex concurrency semantics.

In this chapter, we study an alternative approach to establishing soundness of compiler optimizations under a weak memory model that is easier to use by compiler developers and is well-suited for integration within a certified compiler. The idea is to rely solely on *sequential* reasoning, and our main contribution is a novel sequential (*i.e.*, single-threaded) semantics that can be safely used for analyzing thread-local optimizations under a full-fledged weak memory model.

The proposed approach goes hand in hand with the fact that compiler writers' intuition for thread-local optimizations stems from inspecting sequential code, since, intuitively speaking, non-racy code behaves just like sequential code. In fact, validating optimizations that are correct in sequential programs has been one of the main goals in weak memory models design. Our results provide a formal justification of this intuition, and give grounds for development, verification, and testing of optimizations based on a sequential model.

Example 12. As a concrete simple example, consider an optimization pass that avoids unnecessary reads by locally applying a simplified "store-to-load forwarding" (SLF) as captured by the following pattern:

$$X^{\mathtt{na}} := v \,;\, b := X^{\mathtt{na}} \quad \rightsquigarrow \quad X^{\mathtt{na}} := v \,;\, b := v$$

where X is a shared variable, the `na` superscript denotes non-atomic access to memory, v is an arbitrary value, and b is a thread local register. In sequential programs this transformation is clearly sound. We aim to rely on sequential reasoning for justifying this transformation under weak memory. △

While the idea of using sequential semantics to assist reasoning on concurrent

[1]Informal and pen-and-paper arguments often resulted in detecting miscompilation bugs due to subtle unexpected interaction between language features; see *e.g.*, [1, Remark 7], [49, §2.2], and [51, 52].

programs is not new, our results provide two important advantages. First, while previous work [55, 56, 57] studied a simple concurrency model based on locks or atomic blocks, the current work is the first to realize this idea for a rich weak memory model with a wide spectrum of concurrency features, including atomic accesses of several kinds. In particular, we demonstrate that the proposed sequential semantics is sufficiently expressible to validate certain intricate optimizations of non-atomic accesses across atomics, which are performed by mainstream compilers (as we observed on armv8-a clang 11.0.1 and x86-64 GCC 11.2).

Example 13. Continuing Example 12, a more interesting SLF pass eliminates reads also across other instructions:

$$X^{\mathbf{na}} := v\,;\ \alpha\ ;\, b := X^{\mathbf{na}} \quad \rightsquigarrow \quad X^{\mathbf{na}} := v\,;\ \alpha\ ;\, b := v$$

What patterns of synchronization accesses (composed of C/C++ atomics) may be included in α (besides the fact that it should not contain writes to X) has been a source of confusion before [49, §2.2]. As we show below, the model proposed in this chapter allows one to analyze the soundness of this pass relying solely on a sequential model. △

Second, in contrast to prior work [55, 56, 57], we are *not* targeting a concurrency model based on the "catch-fire" mechanism, which triggers undefined behavior (UB) for data races like in C/C++11 [27]. The practical significance of this choice is that (irrelevant) *load introduction* is a sound program transformation in our model. In contrast, this transformation for non-atomics can never be generally sound in a catch-fire model, since it may introduce data races in the target program that do not exist in the source. Allowing load introduction is necessary to support optimizations based on speculation, which are commonly performed by compilers (clang, in particular), *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.[2] (In fact, the "freeze" instruction recently introduced in LLVM is a tool to support branching on a possibly

[2]See `https://llvm.org/docs/Passes.html` [Accessed Nov-21].

undefined value, which is often a result of load introduction [58].)

Example 14. Consider an optimization pass performing loop invariant code motion (LICM) following the pattern:

$$\textbf{while } B \textbf{ do } \{\, \alpha \,;\, a := X^{\mathtt{na}} \,;\, \beta \,\} \;\rightsquigarrow$$
$$c := X^{\mathtt{na}} \,;\, \textbf{while } B \textbf{ do } \{\, \alpha \,;\, a := c \,;\, \beta \,\}$$

In the case that the loop never executes (when B is false), a possibly racy (irrelevant) load of X is introduced. Thus, this transformation is unsound in catch-fire models. In contrast, we aim to validate such transformations, and again use sequential reasoning for their formal justification (with appropriate restrictions on α and β; see §3.4). △

To demonstrate that sequential reasoning is adequate for validating soundness of optimizations under a weak memory model, we (formally) establish the adequacy of sequential reasoning for verifying optimizations w.r.t. PS2.1 [1]. The latter is a recent version of the "promising semantics", a well-studied model [31, 59, 4, 3] addressing the infamous "out-of-thin-air" problem that admits efficient mapping schemes to modern architectures, as well as several critical programming guarantees. Since this model does not include non-atomic accesses, we extend it with such accesses. In this extension, inspired by LLVM [60], to allow load introduction, which is notoriously hard to support in a relaxed memory model [61], racy non-atomic reads retrieve "undefined" values that can be later "frozen" into any non-deterministic value [58] (rather than invoking UB as in C/C++11).

All in all, the contributions of this chapter are:

1) We develop a sequential model, called SEQ, instrumenting a standard sequential memory with additional mechanisms to reason about program transformations under a weak memory model (§3.2). The sequential model abstracts away complicated interference of other threads, by, in particular, tracking *permissions* to perform non-atomic accesses on certain locations which are non-deterministically gained and dropped with acquire/release atomic accesses. We present *two* notions of behavioral refinement in

SEQ that ensure refinement under weak memory with arbitrary concurrent context: a simple one (§3.2) that suffices for the vast majority of optimizations (including all those involving solely non-atomics), and a more refined one (§3.3) needed for certain transformations involving a non-atomic write and a release/relaxed atomic access. We note that typical programmers need not know about the the sequential model SEQ, which is only needed for compiler developers.

2) We develop a *certified optimizer* for a small C-like concurrent language (§3.4). The optimizer performs four passes of thread-local optimizations: store-to-load forwarding, load-to-load forwarding, dead (overwritten) store elimination, and loop invariant code motion, based on a standard fixpoint analysis in an abstract semantics representing properties of the program executions in SEQ. The optimizer is implemented and certified in Coq. Importantly, its correctness is proved relying *solely* on simulation in SEQ, without any reference to the underlying (much more complex) promising model. Thus, we view this optimizer as a proof of concept demonstrating the applicability of our approach to verify optimization passes, possibly as an extension of CompCert.

3) We extend PS2.1 with non-atomics, with UB for write-write races and undefined value for write-read races (§3.6). All results for PS2.1 in [1] are ported to $\text{PS}^{\textsf{na}}$, the extended model. We believe that $\text{PS}^{\textsf{na}}$ can be useful as a model for an intermediate representation (IR) language, such as LLVM. In turn, defensive programmers may rely on one of the data-race freedom (DRF) guarantees of the model (see [1]), each of which ensures certain stronger and simpler semantics provided that certain races are avoided.

4) We prove an adequacy theorem allowing one to derive the correctness of optimizations in $\text{PS}^{\textsf{na}}$ from (simple or refined) behavioral refinement in SEQ (§3.7).

Our results are fully mechanized in Coq, building on top of the existing formalization of PS2.1. Our development is available at [62].

3.2 The Sequential Permission Machine

We introduce a sequential model, which we denote by SEQ, and present a notion of *behavioral refinement* between sequential programs that we will show to be adequate for reasoning about optimizations of concurrent programs under weak memory consistency: if a target program behaviorally refines a source program, then the source program can be replaced by the target program under any concurrent context assuming weak memory semantics (specifically, PS^{na}).

To understand the intuitions behind SEQ it is important to keep in mind the optimizations we aim to validate. First, since non-atomics are not meant for synchronization, all optimizations allowed in sequential code, including load introduction, should be validated on code involving solely non-atomics. The important exception here is unused store introduction, which is sound in sequential code (although existing compilers avoid this transformation, possibly due to security reasons—we would not want to expose secrets in memory), but trivially unsound in concurrent code, as an unused store of one thread may be read by others.

In contrast, we do *not* aim to allow optimizations on atomic accesses and fences. Understanding optimizations on synchronization code (via atomics) via sequential reasoning is unnatural, and, even if possible, it will significantly complicate our sequential model. Also, since atomic accesses are relatively rare in concurrent programs and often confined in libraries that are manually optimized by experts, the possible performance gain is rather limited. Although these optimizations were extensively studied (especially for C/C++11 [20, 63]), to the best of our knowledge, existing compilers do not perform such optimizations.

Finally, we also aim to allow optimizations of non-atomics *across* atomics. As mentioned in the introduction, these are performed by mainstream compilers and have been a source of confusion before. We will also validate reorderings of relaxed

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

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

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

(CHOICE/RELAXED)
$$\frac{\sigma \xrightarrow{e} \sigma' \quad e \in \{\texttt{choose}(v), \texttt{R}^{\texttt{rlx}}(X,v), \texttt{W}^{\texttt{rlx}}(X,v)\}}{\langle \sigma, P, F, M\rangle \xrightarrow{e} \langle \sigma', P, F, M\rangle}$$

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

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

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

(REL-WRITE)
$$\frac{\sigma \xrightarrow{\texttt{W}^{\texttt{rel}}(X,v)} \sigma' \quad P' \subseteq P \qquad V = M|_P}{\langle \sigma, P, F, M\rangle \xrightarrow{\texttt{W}^{\texttt{rel}}(X,v,P,P',F,V)} \langle \sigma', P', \emptyset, M\rangle}$$

Figure 3.1: Transitions of SEQ

accesses and non-atomics, as well as roach-motel reorderings (one-sided reordering of release/acquire and non-atomics) [64], which are not performed by current mainstream compilers but are naturally supported in SEQ.

Concurrency constructs We assume that shared memory locations are divided into atomic locations ($\mathsf{Loc}^{\mathsf{at}}$) and non-atomic locations ($\mathsf{Loc}^{\mathsf{na}}$), and there is no mixing of atomic and non-atomic accesses to the same location.[3] To simplify the presentation in the paper, we only present a fragment of the model consisting of non-atomics and release/acquire and relaxed reads and writes. Our Coq development includes more features: atomic read-modify-writes (RMWs), release sequences, fences (including sequentially consistent fences), strong relaxed accesses (which do not allow “load buffering” behaviors), and system calls. This covers all C/C++11 features as in [13], except for sequentially consistent accesses which are not supported by the promising semantics.

[3]The problem in supporting such mixing is the LOWER step of PS^{na} that modifies values of outstanding promises. We describe the challenge in §3.7.

Program representation in the paper To keep the presentation abstract, rather than introducing a concrete programming language syntax, we assume that the programming language is represented as a labeled transition system (LTS), with transitions labeled with the action that is performed. Below we use σ to denote the *program state*, which stores the rest of the program to run and the current local register file. Transitions take one of the following forms:

- $\sigma \rightarrow \sigma'$: silent transitions that do not communicate with the memory (*e.g.*, conditionals and local assignments).
- $\sigma \xrightarrow{\texttt{choose}(v)} \sigma'$: transitions resolving a non-deterministic choice (Remark 10 in §3.7 explains why we need to expose these transitions).
- $\sigma \xrightarrow{\texttt{R}^{o_\texttt{R}}(X,v)} \sigma'$ with $o_\texttt{R} \in \{\texttt{na}, \texttt{rlx}, \texttt{acq}\}$: reads value v from location X with mode $o_\texttt{R}$ (non-atomic, relaxed, or acquire).
- $\sigma \xrightarrow{\texttt{W}^{o_\texttt{W}}(X,v)} \sigma'$ with $o_\texttt{W} \in \{\texttt{na}, \texttt{rlx}, \texttt{rel}\}$: writes value v to location X with mode $o_\texttt{W}$ (non-atomic, relaxed, or release).

We assume that programs terminate in states of the form $\sigma = \textbf{return}(v)$, which denote normal termination with v being the final value that is externally observable, or in an "error state", $\sigma = \bot$, denoting UB (*e.g.*, when dividing by 0).

In our examples we use standard code snippets in a toy language to denote programs, and their reading as LTSs should be clear. We also assume a notion of a (sequential) program context $C[\cdot]$, which is a program with a hole allowing one to plug in programs, *e.g.*, $C[X^{\texttt{na}} := 1]$. This notation, which we do not formally specify, is meant for intuitive understanding of our examples. In our Coq formulation, programs are represented using *interaction trees* [65], which provide a convenient formalism supporting this operation.

Values We assume a parametric set Val of values. To support racy non-atomic reads, Val should contain a distinguished element, called "undefined value" and denoted by `undef`. In the refinement notions below, we allow the target program to return any value if the source returns `undef`. For this matter, a partial order $\sqsubseteq$ on Val is defined by: $v \sqsubseteq v' \Leftrightarrow v = v' \vee v' = \texttt{undef}$. This order is lifted to (partial) functions to Val pointwise.

Remark 8. We follow LLVM assuming that branching on `undef` invokes UB. A "freeze" instruction can be used to non-deterministically choose a defined value for `undef`,[4] which is captured by a $\texttt{choose}(v)$ transition in the LTS.

States of SEQ In addition to the current program state σ, each state $S = \langle \sigma, P, F, M \rangle$ of SEQ keeps track of:

- Permission set: a set $P \subseteq \mathsf{Loc}^{\mathsf{na}}$ of non-atomic locations that may be safely accessed. Intuitively, if $X \notin P$, then the access to X is racy.

- Written (non-atomic) locations set: a set $F \subseteq \mathsf{Loc}^{\mathsf{na}}$ of non-atomic locations that were written to by the thread. We track this set in the states (and later use in the definition of behavior refinement) in order to ensure that non-atomic write introduction is unsound in SEQ.

- Memory: a function $M : \mathsf{Loc}^{\mathsf{na}} \to \mathsf{Val}$ assigning a value to every non-atomic location. Since we are not aiming to support optimizations on atomics, there is no need to keep the values of the atomic locations in the memory. (In the refinement notions below, we require that the sequence of atomic accesses generated by the source and the one generated by the target match.)

Transitions of SEQ The transitions are given in Fig. 3.1. Each transition $S \xrightarrow{(e)} S'$ dictates its preconditions (which always require a corresponding program step), the way the different components of the state are updated, and possibly the label e

[4]See `https://llvm.org/docs/LangRef.html#undefined-values` and `https://llvm.org/docs/LangRef.html#freeze-instruction` [Accesses Nov-21].

recorded in the trace when the transition is invoked. The latter is essential for the definition of refinement, which imposes conditions relating the traces (*i.e.*, sequences of transition labels) of SEQ generated by the source program to those generated by the target program.

Concretely, SILENT and CHOICE/RELAXED transitions have no additional preconditions and, except for the program component, do not modify the state. The CHOICE/RELAXED transitions are recorded in the transition label as they need to match in traces of the source and of the target.

Non-atomics are handled differently depending on the permission set: when the program performs a non-atomic read from location X, it loads from the memory if X is in the permission set (NA-READ); or loads `undef` otherwise (RACY-NA-READ). In turn, when the program performs a non-atomic write to X, it writes to memory and adds X to the set of written locations if X is in the permission set (NA-WRITE), or invokes UB (by setting the program state to $\perp$) otherwise (RACY-NA-WRITE). Invoking UB is in accordance with the fact that we aim to invalidate (unused) store introduction. Note that the steps related to non-atomic accesses have no effect on the generated trace, allowing different sequences of non-atomic accesses between the source and the target.

Acquire and release accesses are used for synchronization in the underlying concurrency model. Although they provide more fine grained control than locks, it is helpful to understand an acquire read as a lock acquisition, and a release write as a lock release.[5] In SEQ, these steps non-deterministically update the permission set and the memory, which, intuitively speaking, accounts for any possible interaction with the concurrent environment.

Concretely, ACQ-READ non-deterministically (*i*) gains permissions for some set

[5]Lock acquisition requires an acquire RMW, which is included in our Coq development but elided here to simplify the presentation.

of locations (intuitively, these permissions are acquired from other threads), and (*ii*) gets new values (recorded in a partial function $V : \mathsf{Loc}^{\mathtt{na}} \rightharpoonup \mathsf{Val}$) for the locations in this set. Dually, REL-WRITE non-deterministically loses permissions for some set of locations (intuitively, they are released to other threads). The REL-WRITE transition also resets the written locations set F. Thus, F tracks written non-atomic locations since the last release, which is needed in order to ensure that (possibly racy) writes cannot be introduced after a release, even if the location was written to (by the source) before the release (see Example 21).

In addition, ACQ-READ and REL-WRITE record in the trace (*i.e.*, on their transition labels) the permission set before and after the transition, the written locations set, and the current memory (its updated part in ACQ-READ and "(potentially) released" memory in REL-WRITE). All these are needed for having sufficiently expressive traces that allow us to define an adequate refinement notion.

Behavioral refinement We first define what constitutes a behavior in SEQ.

Definition 15. A *behavior* (in SEQ) is a pair of the form $\langle tr, r\rangle$, where tr is a finite sequence of transition labels, and r is either $\mathtt{trm}(v, F, M)$ denoting normal termination returning v with written flags set F and memory M, $\mathtt{prt}(F)$ denoting a partial (ongoing) execution with current written flags set F, or $\bot$ denoting erroneous termination. We write $S \Downarrow \langle tr, r\rangle$ to mean that a state S *generates* the behavior $\langle tr, r\rangle$, which is inductively defined as follows:

$$\frac{r = \begin{cases}\mathtt{trm}(v, F, M) & \sigma = \mathbf{return}(v)\\ \bot & \sigma = \bot\\ \mathtt{prt}(F) & \text{otherwise}\end{cases}}{\langle \sigma, P, F, M\rangle \Downarrow \langle \epsilon, r\rangle} \qquad \frac{S \to S' \quad S' \Downarrow \langle tr, r\rangle}{S \Downarrow \langle tr, r\rangle} \qquad \frac{S \xrightarrow{e} S' \quad S' \Downarrow \langle tr, r\rangle}{S \Downarrow \langle e \cdot tr, r\rangle}$$

We use standard notations for traces: ϵ for the empty trace, $tr_1 \cdot tr_2$ for appending traces, and $\alpha \in tr$ for occurrence of a label in a trace. We identify a label e with a trace of length one when writing expressions like $e \cdot tr$.

Example 15. For a program state σ that corresponds to $X^{\mathtt{rlx}} := 1\,; Y^{\mathtt{na}} := 2\,;$ $\mathbf{return}(3)$, the state $S = \langle \sigma, P, \emptyset, M\rangle$ with $Y \in P$ has the following behaviors in

SEQ: $\langle \epsilon, \texttt{prt}(\emptyset)\rangle$, $\langle \texttt{W}^{\texttt{rlx}}(X,1), \texttt{prt}(\emptyset)\rangle$, $\langle \texttt{W}^{\texttt{rlx}}(X,1), \texttt{prt}(\{Y\})\rangle$, and the terminating behavior $\langle \texttt{W}^{\texttt{rlx}}(X,1), \texttt{trm}(3,\{Y\}, M[Y \mapsto 2])\rangle$. If $Y \notin P$, then $\langle \texttt{W}^{\texttt{rlx}}(X,1), \bot\rangle$ is the only terminating behavior. $\triangle$

Next, we present the (first) notion of behavioral refinement between programs in SEQ. As standard, one may start by requiring that every behavior $\langle tr_{\text{tgt}}, r_{\text{tgt}}\rangle$ of the target program is also a behavior of the source program. However, to support various optimizations, naive inclusion does not suffice. Instead, we allow the source to generate a matching behavior $\langle tr_{\text{src}}, r_{\text{src}}\rangle$ that is "less committed" than $\langle tr_{\text{tgt}}, r_{\text{tgt}}\rangle$ (denoted $\langle tr_{\text{tgt}}, r_{\text{tgt}}\rangle \sqsubseteq \langle tr_{\text{src}}, r_{\text{src}}\rangle$): the source may return `undef` when the target returns a normal value ($v_{\text{tgt}} \sqsubseteq v_{\text{src}}$), end with "less defined" memory ($M_{\text{tgt}} \sqsubseteq M_{\text{src}}$), and write to more non-atomic locations ($F_{\text{tgt}} \subseteq F_{\text{src}}$). The same holds along the trace: for values recorded in atomic writes, for memories recorded in release writes, and for written locations sets recorded in acquire and release accesses. Finally, UB by the source allows *any* continuation by the target. All these are formally captured by the next definition:

Definition 16. The relation $\sqsubseteq$ on transition labels, traces, and behaviors is given by:

1. Transition labels:

$$\frac{}{e \sqsubseteq e} \qquad \frac{v_{\text{tgt}} \sqsubseteq v_{\text{src}}}{\texttt{W}^{\texttt{rlx}}(X, v_{\text{tgt}}) \sqsubseteq \texttt{W}^{\texttt{rlx}}(X, v_{\text{src}})} \qquad \frac{F_{\text{tgt}} \subseteq F_{\text{src}}}{\texttt{R}^{\texttt{acq}}(X, v, P, P', F_{\text{tgt}}, V) \sqsubseteq \texttt{R}^{\texttt{acq}}(X, v, P, P', F_{\text{src}}, V)}$$

$$\frac{v_{\text{tgt}} \sqsubseteq v_{\text{src}} \qquad F_{\text{tgt}} \subseteq F_{\text{src}} \qquad V_{\text{tgt}} \sqsubseteq V_{\text{src}}}{\texttt{W}^{\texttt{rel}}(X, v_{\text{tgt}}, P, P', F_{\text{tgt}}, V_{\text{tgt}}) \sqsubseteq \texttt{W}^{\texttt{rel}}(X, v_{\text{src}}, P, P', F_{\text{src}}, V_{\text{src}})}$$

2. Traces: $e^1_{\text{tgt}} \cdot \ldots \cdot e^n_{\text{tgt}} \sqsubseteq e^1_{\text{src}} \cdot \ldots \cdot e^n_{\text{src}} \Leftrightarrow \forall k.\ e^k_{\text{tgt}} \sqsubseteq e^k_{\text{src}}$
3. Behaviors:

$$\frac{tr_{\text{tgt}} \sqsubseteq tr_{\text{src}} \qquad v_{\text{tgt}} \sqsubseteq v_{\text{src}} \qquad F_{\text{tgt}} \subseteq F_{\text{src}} \qquad M_{\text{tgt}} \sqsubseteq M_{\text{src}}}{\langle tr_{\text{tgt}}, \texttt{trm}(v_{\text{tgt}}, F_{\text{tgt}}, M_{\text{tgt}})\rangle \sqsubseteq \langle tr_{\text{src}}, \texttt{trm}(v_{\text{src}}, F_{\text{src}}, M_{\text{src}})\rangle}$$

$$\frac{tr_{\text{tgt}} \sqsubseteq tr_{\text{src}} \qquad F_{\text{tgt}} \subseteq F_{\text{src}}}{\langle tr_{\text{tgt}}, \texttt{prt}(F_{\text{tgt}})\rangle \sqsubseteq \langle tr_{\text{src}}, \texttt{prt}(F_{\text{src}})\rangle} \qquad \frac{tr_{\text{tgt}} \sqsubseteq tr_{\text{src}}}{\langle tr_{\text{tgt}} \cdot tr, r\rangle \sqsubseteq \langle tr_{\text{src}}, \bot\rangle}$$

Definition 17. We write $S_{\text{tgt}} \sqsubseteq S_{\text{src}}$ if $S_{\text{tgt}} \Downarrow \langle tr_{\text{tgt}}, r_{\text{tgt}} \rangle$ implies that $S_{\text{src}} \Downarrow \langle tr_{\text{src}}, r_{\text{src}} \rangle$ for some behavior $\langle tr_{\text{src}}, r_{\text{src}} \rangle$ such that $\langle tr_{\text{tgt}}, r_{\text{tgt}} \rangle \sqsubseteq \langle tr_{\text{src}}, r_{\text{src}} \rangle$. A program state σ_{tgt} *behaviorally refines* a program state σ_{src}, denoted by $\sigma_{\text{tgt}} \sqsubseteq \sigma_{\text{src}}$, if $\langle \sigma_{\text{tgt}}, P, F, M \rangle \sqsubseteq \langle \sigma_{\text{src}}, P, F, M \rangle$ for every P, F, M.

Next, we present a sequence of examples demonstrating several subtleties in the above definitions. In these examples, when writing $prog_{\text{src}} \leadsto prog_{\text{tgt}}$ for two code snippets $prog_{\text{src}}$ and $prog_{\text{tgt}}$, we mean that for any (sequential) context C, the state σ_{tgt} that corresponds to $C[prog_{\text{tgt}}]$ (with some initial register file) behaviorally refines the state σ_{src} that runs $C[prog_{\text{src}}]$ (with the same initial register file). We write $prog_{\text{src}} \not\leadsto prog_{\text{tgt}}$ for the negation of $prog_{\text{src}} \leadsto prog_{\text{tgt}}$.

Remark 9. Our results that are formalized in Coq also address reasoning about program transformations in SEQ. In particular, they allow one to lift local refinement properties (such as the ones listed in the examples below) to any sequential context C. Concretely, we define a simulation relation between SEQ states and prove that it admits certain congruence properties. Then, by establishing simulation between $prog_{\text{src}}$ and $prog_{\text{tgt}}$, we can derive $prog_{\text{src}} \leadsto prog_{\text{tgt}}$ as defined above. Since these techniques are fairly standard in compiler verification, and our main focus is to reduce reasoning about concurrent code to reasoning about sequential code, we omit these details from the paper. In the example below, we intend to give the right intuitions, rather than precise refinement arguments.

Example 16 (Reordering of non-atomics)**.** Non-atomic accesses to different locations can be freely reordered in SEQ, *e.g.*, $a := X^{\texttt{na}} \,;\, Y^{\texttt{na}} := v \leadsto Y^{\texttt{na}} := v \,;\, a := X^{\texttt{na}}$ where $X \neq Y$. Consider a general context C, and let σ_{src} and σ_{tgt} be the program states that correspond to $C[a := X^{\texttt{na}} \,;\, Y^{\texttt{na}} := v]$ and $C[Y^{\texttt{na}} := v \,;\, a := X^{\texttt{na}}]$, respectively, with the same initial register file. Suppose that $\langle \sigma_{\text{tgt}}, P, F, M \rangle \Downarrow \langle tr^{\text{tgt}}, r^{\text{tgt}} \rangle$. Then, by executing two steps in the source (a read from X followed by a write to Y) at the time the target executes its write to Y, one can show that $\langle \sigma_{\text{src}}, P, F, M \rangle \Downarrow \langle tr^{\text{tgt}}, r^{\text{tgt}} \rangle$. In particular, after the target performs the read, the source and the target reach the same program state.

On the other hand, reordering of non-atomics to the *same* location is disallowed, *e.g.*, the reordering of a load followed by a store $a := X^{\texttt{na}} \,;\, X^{\texttt{na}} := 1 \not\leadsto X^{\texttt{na}} := 1 \,;\, a := X^{\texttt{na}}$. Indeed, for the context $C = \cdot \,;\, \textbf{return}(a)$, we have $\langle \sigma_{\text{tgt}}, P, F, M \rangle \Downarrow \langle \epsilon, \texttt{trm}(1, F \cup \{X\}, M[X \mapsto$ starting from a state with $X \in P$ and $M(X) = 2$. However, the only terminating behavior that is generated by the source program from this state is $\langle \epsilon, \texttt{trm}(2, F \cup \{X\}, M[X \mapsto 1]) \rangle$.

$\triangle$

Example 17 (Eliminations of non-atomics)**.** Behavioral refinement holds for the following pairs, in which a non-atomic access is eliminated:

$$\begin{array}{cc}
(i) & X^{\mathrm{na}} := v\,;\, X^{\mathrm{na}} := v' \rightsquigarrow X^{\mathrm{na}} := v' \\
(ii) & X^{\mathrm{na}} := v\,;\, a := X^{\mathrm{na}} \rightsquigarrow X^{\mathrm{na}} := v\,;\, a := v \\
(iii) & a := X^{\mathrm{na}}\,;\, b := X^{\mathrm{na}} \rightsquigarrow a := X^{\mathrm{na}}\,;\, b := a \\
(iv) & a := X^{\mathrm{na}}\,;\, X^{\mathrm{na}} := a \rightsquigarrow a := X^{\mathrm{na}}
\end{array}$$

Note that for the read-before-write elimination ((*iv*) above), the written locations set in the final state of the source may be a strict superset of the one of the target ($F_{\mathrm{tgt}} \subset F_{\mathrm{src}}$), which is allowed in the definition above. Conceptually, the optimized program may avoid writes to some locations that are performed by the source.

In contrast, the introduction of a write after a read is unsound due to the conditions on the written locations set F. For example:

$$a := X^{\mathrm{na}}\,;\, \textbf{if } a \neq v \textbf{ then } X^{\mathrm{na}} := v \quad \not\rightsquigarrow \quad a := X^{\mathrm{na}}\,;\, X^{\mathrm{na}} := v$$

In this example, starting from $F = \emptyset$ and permission to access X, the target ends its execution with $F_{\mathrm{tgt}} = \{X\}$, while the source has $F_{\mathrm{src}} = \emptyset$.

In turn, the other introductions of non-atomics obtained as converses of (*i*)-(*iii*) above provide additional instances of refinements in SEQ. This intuitively corresponds to the fact that non-atomics are cannot induce synchronization. △

Example 18 (Reordering across loops)**.** Reordering a non-atomic write before a possibly infinite local computation is unsound, as it introduces a write if the loop never terminates:

$$\textbf{while } (...) \textbf{ do } \{...\}\,;\, X^{\mathrm{na}} := v \quad \not\rightsquigarrow \quad X^{\mathrm{na}} := v\,;\, \textbf{while } (...) \textbf{ do } \{...\}$$

In SEQ, when starting without permission on X ($X \notin P$), the target program generates the behavior $\langle \epsilon, \bot \rangle$, but the source may not be able to generate a matching behavior if the loop is not terminating.

A variant of this example demonstrates why we have to match *partial* traces with the condition that $F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}}$:

$$\begin{array}{lcl}
a := X^{\mathrm{na}}\,; & & a := X^{\mathrm{na}}\,; \\
\textbf{if } a \neq 1 \textbf{ then } X^{\mathrm{na}} := 1\,; & \not\rightsquigarrow & \textbf{if } a \neq 1 \textbf{ then } X^{\mathrm{na}} := 1\,; \\
\textbf{while } (...) \textbf{ do } \{...\}\,;\, X^{\mathrm{na}} := 2 & & X^{\mathrm{na}} := 2\,;\, \textbf{while } (...) \textbf{ do } \{...\}
\end{array}$$

If the target invokes UB and generates $\langle \epsilon, \bot \rangle$, then it must be the case that we started without permission on X, and the source may invoke UB and generate $\langle \epsilon, \bot \rangle$ as well. Thus, in order to obtain a behavior of the target that is not matched by a behavior of the source, in case the loop is non-terminating, we must consider behaviors before termination $\langle \_, \texttt{prt}(F)\rangle$. Indeed, when starting with permission on X and $M(X) = 1$, the target generates $\langle \epsilon, \texttt{prt}(\{X\})\rangle$, but, if the loop does not terminate, the only behavior of the source is $\langle \epsilon, \emptyset\rangle$.

In contrast, reads may be reordered with possibly non-terminating local computation:

$$\textbf{while } (...) \textbf{ do } \{...\}\,;\, a := X^{\texttt{na}} \quad \rightsquigarrow \quad a := X^{\texttt{na}}\,;\, \textbf{while } (...) \textbf{ do } \{...\}$$

Indeed, in partial traces only the written locations set F has to match, and this set is the same in executions of the two programs. △

Example 19 (Unused load elimination and introduction)**.** The transformations that eliminate/introduce an unused load $a := X^{\texttt{na}} \rightsquigarrow \textbf{skip}$ and $\textbf{skip} \rightsquigarrow a := X^{\texttt{na}}$ trivially correspond to behavioral refinements in SEQ. For the latter, we need that a non-atomic read without permission does not invoke UB. We note that in the paper presentation, we have to assume that a does not occur in the context. Nevertheless, our Coq formalization that uses interaction trees can easily express a computation that reads a value and does not return the result to its continuation. This computation is interchangeable with a no-op under *any* context. △

Example 20 (Reordering of atomics and non-atomics)**.** Reordering of atomic and non-atomic accesses follows the "roach-motel" principle. The following are forbidden:

$$\begin{array}{lrcl}
(i) & a := X^{\texttt{acq}}\,;\, Y^{\texttt{na}} := v & \not\rightsquigarrow & Y^{\texttt{na}} := v\,;\, a := X^{\texttt{acq}} \\
(ii) & Y^{\texttt{na}} := v'\,;\, X^{\texttt{rel}} := v & \not\rightsquigarrow & X^{\texttt{rel}} := v\,;\, Y^{\texttt{na}} := v' \\
(iii) & a := X^{\texttt{acq}}\,;\, b := Y^{\texttt{na}} & \not\rightsquigarrow & b := Y^{\texttt{na}}\,;\, a := X^{\texttt{acq}} \\
(iv) & a := Y^{\texttt{na}}\,;\, X^{\texttt{rel}} := v & \not\rightsquigarrow & X^{\texttt{rel}} := v\,;\, a := Y^{\texttt{na}}
\end{array}$$

In (i), starting without permission on Y ($Y \notin P$), the target invokes UB, thus generating the behavior $\langle \epsilon, \bot\rangle$. The source, however, has to perform the acquire read before invoking UB, thus generating terminating behaviors of the form $\langle \texttt{R}^{\texttt{acq}}(\_), \bot\rangle$ only.

In (ii), if the release write to X loses the permission on Y (transitioning from a state with $Y \in P$ to a state with $Y \notin P$), then the target program invokes UB, generating a behavior of the form $\langle \_, \bot\rangle$. However, since we start with $Y \in P$, the source does not invoke UB, and cannot generate any behavior of the form $\langle \_, \bot\rangle$.

In (iii), for the context $C = \cdot\,;\, \textbf{return}(b)$, if a permission on Y is gained by the acquire read from X, and we start with $Y \notin P$ (and $M(Y) \neq \texttt{undef}$), the target generates a behavior of the form $\langle \texttt{R}^{\texttt{acq}}(X, \_, P, P \cup \{Y\}, \_, \_), \texttt{trm}(\texttt{undef}, \_, \_)\rangle$, whereas the source cannot perform racy read on Y.

In (iv), for the context $C = \cdot\,;\, \textbf{return}(a)$, if we start with $Y \in P$, and this permission is lost by the release write, the target generates a behavior of the form $\langle \_, \texttt{trm}(\texttt{undef}, \_, \_)\rangle$, whereas the source cannot perform racy read on Y at all.

Next, the following converses of the above are validated:

$$\begin{array}{ll}(i') & Y^{\texttt{na}} := v\,;\, a := X^{\texttt{acq}} \rightsquigarrow a := X^{\texttt{acq}}\,;\, Y^{\texttt{na}} := v \\ (iii') & b := Y^{\texttt{na}}\,;\, a := X^{\texttt{acq}} \rightsquigarrow a := X^{\texttt{acq}}\,;\, b := Y^{\texttt{na}} \\ (iv') & X^{\texttt{rel}} := v\,;\, a := Y^{\texttt{na}} \rightsquigarrow a := Y^{\texttt{na}}\,;\, X^{\texttt{rel}} := v\end{array}$$

For (i') we use the fact that acquire transitions of the target can be matched by acquire transitions of the source annotated with $F_{\text{tgt}} \subseteq F_{\text{src}}$ (since we may have $Y \in F_{\text{src}}$ but not $Y \in F_{\text{tgt}}$ when performing the acquire), as well as the fact that the source may invoke UB earlier than the target (in particular, $\langle \texttt{R}^{\texttt{acq}}(\_), \bot\rangle \sqsubseteq \langle \epsilon, \bot\rangle$). In turn, (iii') and (iv') demonstrate the need in allowing the source to return `undef` when the target returns a defined value. This is needed, for instance, if the acquire read in (iii') gains permission on Y.

Finally, despite being a valid roach-motel reordering, the converse of (ii) is disallowed by the current behavior refinement. It is supported by the more refined notion in §3.3. △

Example 21. Stores cannot be introduced even if they already occur before a release write:

$$X^{\texttt{na}} := v\,;\, Y^{\texttt{rel}} := 1 \quad \not\rightsquigarrow \quad X^{\texttt{na}} := v\,;\, Y^{\texttt{rel}} := 1\,;\, X^{\texttt{na}} := v$$

Intuitively, if a write is protected by a lock, another one should not be introduced after the lock is released. Formally, since release writes reset the written locations set, the target's terminating behavior has $X \in F$, while the source ends with $F = \emptyset$. In contrast, the transformation is validated with `rlx` instead of `rel` above. △

Example 22 (Store-to-load forwarding across atomics)**.** Reads can be eliminated after writes *across atomics*:

$$X^{\texttt{na}} := v\,;\, \alpha\,;\, b := X^{\texttt{na}} \quad \rightsquigarrow \quad X^{\texttt{na}} := v\,;\, \alpha\,;\, b := v$$

where $\alpha \in \{a := Y^{\texttt{rlx}}, Y^{\texttt{rlx}} := v', a := Y^{\texttt{acq}}, Y^{\texttt{rel}} := v'\}$. If we start with $X \notin P$, the source raises UB, and generates $\langle \epsilon, \bot\rangle$, which matches any target behavior. Otherwise, the relevant write step of the source sets $M(X) = v$, and $M(X)$ is not altered by α (in particular, it is important here that an acquire read can only modify values of locations that gained permission). Thus, either v is read by $b := X^{\texttt{na}}$, or, if α corresponds to a release write that transferred the permission on X, the source will read `undef`, and we have $v \sqsubseteq$ `undef`. In any case, the source matches every behavior of the target. △

Example 23. Reads *cannot* be eliminated after writes across *release-acquire pairs*:

$$X^{\texttt{na}} := v\,;\, Y^{\texttt{rel}} := v'\,;\, a := Z^{\texttt{acq}}\,;\, b := X^{\texttt{na}} \quad \not\rightsquigarrow$$
$$X^{\texttt{na}} := v\,;\, Y^{\texttt{rel}} := v'\,;\, a := Z^{\texttt{acq}}\,;\, b := v$$

Intuitively, another thread may safely access X between the release and the acquire and change its value. To see this in SEQ, consider the execution of the target program when permission to X is lost by the release write, and regained by the acquire read. The updated portion of the memory V, including a new (non-deterministic) value for X, is recorded on the acquire transition in the trace. To match the behavior of the target, the source program has to have the same updated memory in its acquire transition, and when $V(X) \neq v$, the source will not be able to later read v from X. This example demonstrates the need in updating the values in memory (for locations that gained permission) in acquire steps. $\triangle$

3.3 Advanced Behavior Refinement

As we show in §3.7, the above notion of behavioral refinement in SEQ is adequate for reasoning about optimizations in the promising semantics. As shown above, it is also precise enough to verify a variety of optimizations. However, optimizations including both an atomic access and a non-atomic write are beyond its power: although they are meant to be sound (and they are sound in the promising semantics), the above notion invalidates them. In this section, we discuss this issue that stems from two different reasons, and then present a more refined notion of behavioral refinement (implied by the simple one above) that addresses this challenge. We note that, since our result in §3.7 provides contextual refinement, one may mix and match—prove most optimization passes using the simple notion in §3.2, and use the one of this section for several more involved program transformations.

Late UB A simple example of a sound optimization that is invalidated by the above notion is the following:

$$a := X^{\texttt{rlx}} \,;\, Y^{\texttt{na}} := v \quad \rightsquigarrow \quad Y^{\texttt{na}} := v \,;\, a := X^{\texttt{rlx}}$$

Indeed, the reasoning in Example 20 (i) that shows why an acquire read followed by a non-atomic write cannot be reordered applies as is in this case as well: starting without permission on Y $(Y \notin P)$, the target program invokes UB, thus generating the behavior $\langle \epsilon, \bot \rangle$. The source, however, has to perform the relaxed read before invoking UB, thus generating terminating behaviors $\langle tr_{\text{src}}, \bot \rangle$ with tr_{src} consisting of a $\mathtt{R}^{\mathtt{rlx}}$ label. Intuitively, however, this should not matter since the source program anyway invokes UB, in which case the target's behavior is immaterial. Thus, we would like to allow to match any behavior $\langle tr_{\text{tgt}}, r_{\text{tgt}} \rangle$ of the target program by *any UB behavior* $\langle tr_{\text{src}}, \bot \rangle$ of the source. Nevertheless, for two reasons, this solution requires extra care.

First, it essentially allows reordering of any access with an operation that invokes UB, *e.g.*, $\alpha\,;\, a := {}^1\!/\!_0 \leadsto a := {}^1\!/\!_0\,;\, \alpha$. As the next example shows, in concurrent settings this reordering must be invalidated if α contains an acquire read.

Example 24. Consider the following optimizations:

$$\begin{array}{l} a := X^{\mathtt{rlx}}\,; \\ \textbf{if } a = 1 \textbf{ then} \\ \quad a := X^{\mathtt{acq}}\,; \\ \quad b := {}^1\!/\!_0 \\ \textbf{else } Y^{\mathtt{rlx}} := 1 \end{array} \quad \leadsto \quad \begin{array}{l} a := X^{\mathtt{rlx}}\,; \\ \textbf{if } a = 1 \textbf{ then} \\ \quad b := {}^1\!/\!_0\,; \\ \quad a := X^{\mathtt{acq}} \\ \textbf{else } Y^{\mathtt{rlx}} := 1 \end{array} \quad \leadsto \ldots \leadsto \quad \begin{array}{l} Y^{\mathtt{rlx}} := 1\,; \\ a := X^{\mathtt{rlx}}\,; \\ \textbf{if } a = 1 \textbf{ then} \\ \quad b := {}^1\!/\!_0\,; \\ \quad a := X^{\mathtt{acq}} \end{array}$$

First, we perform the (unsound) reordering of an acquire read with UB-invoking operation. Then, a sequence of standard optimizations (start with $b := {}^1\!/\!_0 \leadsto Y^{\mathtt{rlx}} := 1\,;\, b := {}^1\!/\!_0$, then hoist $Y^{\mathtt{rlx}} := 1$ from both branches of the conditional and reorder it with $a := X^{\mathtt{rlx}}$) lead to the program on the right. Now, if the concurrent context consists of another thread with the code: $c := Y^{\mathtt{rlx}}\,;\, X^{\mathtt{rel}} := c$, then UB is possible for the target, but not for the source. △

Thus, to keep invalidating the reordering of an acquire operation followed by UB, we require that there are not any acquire accesses in the suffix of the source trace tr_{src} (in the source's path towards UB) that does not match the target's trace tr_{tgt}. For instance, this invalidates the transformations $a := X^{\mathtt{acq}}\,;\, b := {}^1\!/\!_0 \leadsto b := {}^1\!/\!_0\,;\, a := X^{\mathtt{acq}}$ and $a := X^{\mathtt{acq}}\,;\, Y^{\mathtt{na}} := v \leadsto Y^{\mathtt{na}} := v\,;\, a := X^{\mathtt{acq}}$: if we start without permission on Y, the target's behavior $\langle \epsilon, \bot \rangle$ does not match any source behavior.

Second, it is crucial to make sure that for executing this suffix, the source does not make assumptions on the environment. To see this, consider the following example:

$$\begin{array}{l} a := X^{\texttt{rlx}}\,;\ \textbf{if}\ a = 1\ \textbf{then}\ b := 1/0\,; \\ \textbf{while}\ (...)\ \textbf{do}\ \{...\} \end{array} \quad \not\rightsquigarrow \quad \begin{array}{l} b := 1/0\,;\ a := X^{\texttt{rlx}}\,; \\ \textbf{while}\ (...)\ \textbf{do}\ \{...\} \end{array}$$

Without additional restrictions, since we allow any behavior of the target to be matched with $\langle \texttt{R}^{\texttt{rlx}}(X, 1), \bot\rangle$ of the source, this transformation, which is clearly unsound (even in sequential programs), will be validated by SEQ. Intuitively speaking, what went wrong here is that the source matches the UB of the target by reading 1 from X, whereas a concurrent environment may not provide this option.

To address this issue, when we match the UB of the target by a suffix of source trace that leads to UB we need to make sure that the source avoids making assumptions on the concurrent environment. Technically, we achieve this by assuming that read values of atomic reads, permission gains and losses, and memory updates (V on acquire transitions) are dictated by an *oracle*, which intuitively represents a possible concurrent environment. We then require that behavioral refinement holds for *any oracle* (which has to satisfy certain progress and monotonicity conditions that any environment satisfies). In particular, in the example above, the source has to match the target's UB also for an oracle that forces the source to read $X \neq 1$, in which case the source cannot invoke UB. In contrast, in the earlier example for the need in late UB ($a := X^{\texttt{rlx}}\,;\ Y^{\texttt{na}} := v \quad\rightsquigarrow\quad Y^{\texttt{na}} := v\,;\ a := X^{\texttt{rlx}}$), if we start without permission on Y, the source invokes UB for any oracle as its write is independent of the read.

Writes across release Roach motel reordering of a release write followed by a non-atomic write pose an additional challenge to sequential reasoning:

$$X^{\texttt{rel}} := v\,;\ Y^{\texttt{na}} := v' \quad\rightsquigarrow\quad Y^{\texttt{na}} := v'\,;\ X^{\texttt{rel}} := v$$

Even if we modify behavioral refinement as discussed above, some behaviors of the target are not matched by the source. Concretely, starting with permission on Y, the

(BEH-TERMINAL)

$$\frac{v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}} \qquad F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}} \qquad M_{\mathrm{tgt}} \sqsubseteq M_{\mathrm{src}}}{\langle \epsilon, \mathtt{trm}(v_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}})\rangle \sqsubseteq_R \langle \epsilon, \mathtt{trm}(v_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}})\rangle}$$

(BEH-RLX)

$$\frac{\begin{array}{c} e_{\mathrm{tgt}} \sqsubseteq e_{\mathrm{src}} \qquad \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \\ e_{\mathrm{tgt}} = \mathtt{R}^{\mathtt{rlx}}(\_,\_) \lor e_{\mathrm{tgt}} = \mathtt{W}^{\mathtt{rlx}}(\_,\_) \end{array}}{\langle e_{\mathrm{tgt}} \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle e_{\mathrm{src}} \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle}$$

(BEH-PARTIAL)

$$\frac{\mathtt{R}^{\mathtt{acq}}(\_) \notin tr_{\mathrm{src}} \qquad F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}} \cup \bigcup\{F \mid \mathtt{W}^{\mathtt{rel}}(\_,\_,\_,\_,F,\_) \in tr_{\mathrm{src}}\}}{\langle \epsilon, \mathtt{prt}(F_{\mathrm{tgt}})\rangle \sqsubseteq_R \langle tr_{\mathrm{src}}, \mathtt{prt}(F_{\mathrm{src}})\rangle}$$

(BEH-FAILURE)

$$\frac{\mathtt{R}^{\mathtt{acq}}(\_) \notin tr_{\mathrm{src}}}{\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle tr_{\mathrm{src}}, \bot\rangle}$$

(BEH-ACQ-READ)

$$\frac{\begin{array}{c} \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_{\emptyset} \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \\ F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}} \end{array}}{\begin{array}{c} \langle \mathtt{R}^{\mathtt{acq}}(X, v, P, P', F_{\mathrm{tgt}}, V) \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \\ \langle \mathtt{R}^{\mathtt{acq}}(X, v, P, P', F_{\mathrm{src}}, V) \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \end{array}}$$

(BEH-REL-WRITE)

$$\frac{\begin{array}{c} v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}} \qquad \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_{R'} \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \\ R' = (R \setminus F_{\mathrm{src}}) \cup (F_{\mathrm{tgt}} \setminus F_{\mathrm{src}}) \cup \{Y \in \mathsf{Loc}^{\mathtt{na}} \mid V_{\mathrm{tgt}}(Y) \not\sqsubseteq V_{\mathrm{src}}(Y)\} \end{array}}{\begin{array}{c} \langle \mathtt{W}^{\mathtt{rel}}(X, v_{\mathrm{tgt}}, P, P', F_{\mathrm{tgt}}, V_{\mathrm{tgt}}) \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \\ \langle \mathtt{W}^{\mathtt{rel}}(X, v_{\mathrm{src}}, P, P', F_{\mathrm{src}}, V_{\mathrm{src}}) \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \end{array}}$$

Figure 3.2: Behavioral refinement up to a commitment set $R \subseteq \mathsf{Loc}^{\mathtt{na}}$

target's label $\mathtt{W}^{\mathtt{rel}}(X, v, P, P', F_{\mathrm{tgt}}, V_{\mathrm{tgt}})$ must have $Y \in F_{\mathrm{tgt}}$ and $V_{\mathrm{tgt}}(Y) = v'$, whereas the source is confined by the initial state (which may have $Y \notin F_{\mathrm{src}}$ or $V_{\mathrm{src}}(Y) \neq v'$). Intuitively, this should not be a problem: the source is going to write to Y after the release, and other threads can observe that write.

To solve this, we need to allow the source to generate release labels with different written locations set and memory compared to the target's labels, provided that later on the source will write to the non-atomic locations that were different. Technically, we achieve this by parameterizing behavioral refinement with a "commitment set" R, which is a set of non-atomic locations that the source program must write to before it terminates or executes an acquire read. (Fulfilling commitments *after* an acquire read corresponds to the disallowed reordering of writes after an acquire read.) Initially, the commitment set is empty. Then, we modify (remove fulfilled commitments and add new ones) this set with every release transition. In the end of the execution and with every acquire transition, we verify that all commitments were fulfilled. Finally, the non-terminating behaviors $\langle tr, \mathtt{prt}(F)\rangle$ should allow the source program to continue its execution and fulfill the outstanding commitments.

Advanced behavior refinement The above solutions are formalized as follows. First, when checking for refinement between a source program and a target program in SEQ, we use an *oracle* to represent the environment of the thread. To pass only relevant information to the oracle, we use the following notation for stripping transition labels (extended pointwise to traces):

$$|e| \triangleq \begin{cases} \mathtt{R}^{\mathtt{acq}}(X, v, P, P', V) & \text{for } e = \mathtt{R}^{\mathtt{acq}}(X, v, P, P', F, V) \\ \mathtt{W}^{\mathtt{rel}}(X, v, P, P') & \text{for } e = \mathtt{W}^{\mathtt{rel}}(X, v, P, P', F, V) \\ e & \text{otherwise} \end{cases}$$

Definition 18. An oracle Ω is an LTS over stripped transition labels such that the following hold:

- Progress: In every state w of Ω and for every $X \in \mathsf{Loc}^{\mathtt{at}}$, $v \in \mathsf{Val}$, and $P \subseteq \mathsf{Loc}^{\mathtt{na}}$, transitions $\mathtt{choose}(\_)$, $\mathtt{R}^{\mathtt{rlx}}(X, \_)$, $\mathtt{W}^{\mathtt{rlx}}(X, v)$, $\mathtt{R}^{\mathtt{acq}}(X, \_, P, \_, \_)$, and $\mathtt{W}^{\mathtt{rel}}(X, v, P, \_)$ are enabled for some (valid) values of "_".
- Monotonicity: If $w \xrightarrow{e}_{\Omega} w'$ and $e \sqsubseteq e'$, then $w \xrightarrow{e'}_{\Omega} w'$.

The progress condition allows the source to continue its execution and fulfill its commitments after the target has terminated. Monotonicity is required to allow the refinement of `undef` in the source by any defined value. We say that a trace tr is *allowed* by an oracle Ω, denoted by $tr \in \mathsf{Tr}(\Omega)$, if $|tr|$ is a trace of Ω (*i.e.*, a sequence of symbols that Ω can execute, starting from its initial state).

Next, the notion of a behavioral refinement up to a commitment set is formulated in Fig. 3.2. It modifies the one in Def. 16 by allowing the source to invoke UB later than the target (in BEH-FAILURE), while tracking and checking the commitment set R. Each time a release write $\mathtt{W}^{\mathtt{rel}}(X, v, P, P', F_{\text{tgt}}, V_{\text{tgt}})$ is added to the target's trace, we compare it to the matching one by the source $\mathtt{W}^{\mathtt{rel}}(X, v, P, P', F_{\text{src}}, V_{\text{src}})$, and set the new commitment set R' to consist of the locations Y that: (i) were written to by the target but not by the source ($Y \in F_{\text{tgt}} \setminus F_{\text{src}}$); (ii) have a value in the target memory that does not refine the value of the source ($V_{\text{tgt}}(Y) \not\sqsubseteq V_{\text{src}}(Y)$); or (iii) were included in the previous commitment set and not written yet by the source ($Y \in R \setminus F_{\text{src}}$). Upon termination or acquire read, in addition to $F_{\text{tgt}} \subseteq F_{\text{src}}$, we require that all outstanding commitments were fulfilled by the source ($R \subseteq F_{\text{src}}$). Finally, refinement of non-terminating behaviors BEH-PARTIAL allow the source to take more steps (but not acquire reads) for fulfilling its commitments. Since the written locations set is reset with every release write, to see what locations the source has written to, we add all F sets in the release operations in the source's trace to those in the final F set.

With the above definition, the more refined behavioral refinement notion is stated as follows:

Definition 19. A program state σ_{tgt} *weakly behaviorally refines* a program state σ_{src}, denoted by $\sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$, if for every oracle Ω, if $\langle \sigma_{\text{tgt}}, P, F, M \rangle \Downarrow \langle tr_{\text{tgt}}, r_{\text{tgt}} \rangle$ and $tr_{\text{tgt}} \in \mathsf{Tr}(\Omega)$, then $\langle \sigma_{\text{src}}, P, F, M \rangle \Downarrow \langle tr_{\text{src}}, r_{\text{src}} \rangle$ for some $\langle tr_{\text{src}}, r_{\text{src}} \rangle$ such that $\langle tr_{\text{tgt}}, r_{\text{tgt}} \rangle \sqsubseteq_{\emptyset} \langle tr_{\text{src}}, r_{\text{src}} \rangle$ and $tr_{\text{src}} \in \mathsf{Tr}(\Omega)$.

Proposition 7. $\sigma_{\text{tgt}} \sqsubseteq \sigma_{\text{src}} \Rightarrow \sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$.

Example 25 (Overwritten store elimination across atomics)**.** Consider the elimination of a write after another write to the same location across an atomic access:

$$X^{\texttt{na}} := v \,;\; \boxed{\alpha} \,;\; X^{\texttt{na}} := v' \quad \rightsquigarrow \quad \boxed{\alpha} \,;\; X^{\texttt{na}} := v'$$

where $\boxed{\alpha} \in \{b := Y^{\texttt{rlx}}, Y^{\texttt{rlx}} := v_Y, b := Y^{\texttt{acq}}, Y^{\texttt{rel}} := v_Y\}$. The three cases except for $\boxed{\alpha} = Y^{\texttt{rel}} := v_Y$ are easily validated by the simple behavioral refinement in SEQ (here it is needed that the source may have larger F sets).

The case that $\boxed{\alpha}$ is a release write should be also considered sound,[6] since, roughly speaking, other threads that can observe $X^{\texttt{na}} := v$ can always also observe $X^{\texttt{na}} := v'$ instead. (In particular, this optimization is sound in the promising semantics.) Nevertheless, the simple refinement notion in §3.2 invalidates this optimization ($\sigma_{\text{tgt}} \not\sqsubseteq \sigma_{\text{src}}$): starting with permission on X, the memory recorded in release writes in the source is confined to have $M(X) = v$, while the target has the value of the initial memory. In turn, we do have $\sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$. In particular, consider the empty context, and let $\texttt{rel}(P, P', F, u) \triangleq \texttt{W}^{\texttt{rel}}(Y, v_Y, P, P', F, M[X \mapsto u])$. If we start with permission on X and do not release it, then for $r = \texttt{trm}(\texttt{unit}, \{X\}, M[X \mapsto v'])$,

$$\langle \texttt{rel}(\{X\}, \{X\}, \{X\}, v), r \rangle \sqsubseteq_{\emptyset} \langle \texttt{rel}(\{X\}, \{X\}, \emptyset, M(X)), r \rangle$$

follows from $\langle \epsilon, r \rangle \sqsubseteq_{\{X\}} \langle \epsilon, r \rangle$. If we start with permission on X and release it, then

$$\langle \texttt{rel}(\{X\}, \emptyset, \{X\}, v), \bot \rangle \sqsubseteq_{\emptyset} \langle \texttt{rel}(\{X\}, \emptyset, \emptyset, M(X)), \bot \rangle$$

[6]Currently, it is not performed by mainstream compilers (checked for armv8-a clang 11.0.1 and x86-64 GCC 11.2).

follows from $\langle \epsilon, \bot \rangle \sqsubseteq_{\{X\}} \langle \epsilon, \bot \rangle$. If we start without permission on X, then, using BEH-FAILURE, we have:

$$\langle \texttt{rel}(\emptyset, \emptyset, \{X\}, v), \bot \rangle \sqsubseteq_{\emptyset} \langle \epsilon, \bot \rangle. \qquad \triangle$$

3.4 An Overview of a Certified Optimizer

We implemented in Coq a verified optimizer that optimizes an arbitrary program written in WHILE, a simple C-like language, that is interpreted as an interaction trees program, for which our adequacy theorem in §3.7 is stated. The optimizer's correctness proof relies solely on SEQ, thus showcasing the applicability of SEQ for compiler verification.[7]

The optimizer statically analyzes a given sequential program by performing a fixpoint computation in an abstract semantics and optimizes the program based on the static analysis. Generally speaking, the analysis result assigns predicates on states of SEQ to each program point. Using the analysis result, the optimizer transforms the program, for instance, a non-atomic read from X into a register assignment if the analysis ensures that X has certain value.

The optimization process consists of four optimization passes, store-to-load forwarding (SLF), load-to-load forwarding (LLF), dead (overwritten) store elimination (DSE), and loop invariant code motion (LICM), which, on the memory trace level, are captured as follows:

SLF	$X^{\text{na}} := v\,;\ \alpha\ ;\ b := X^{\text{na}} \rightsquigarrow X^{\text{na}} := v\,;\ \alpha\ ;\ b := v$
LLF	$a := X^{\text{na}}\,;\ \beta\ ;\ b := X^{\text{na}} \rightsquigarrow a := X^{\text{na}}\,;\ \beta\ ;\ b := a$
DSE	$X^{\text{na}} := a\,;\ \gamma\ ;\ X^{\text{na}} := b \rightsquigarrow \textbf{skip}\,;\ \gamma\ ;\ X^{\text{na}} := b$
LICM	$\textbf{while}\ (...)\ \textbf{do}\ \{\ \beta_1\ ;\ a := X^{\text{na}}\,;\ \beta_2\ \} \rightsquigarrow$ $c := X^{\text{na}}\,;\ \textbf{while}\ (...)\ \textbf{do}\ \{\ \beta_1\ ;\ a := c\,;\ \beta_2\ \}$

where α contains no writes to X or release-acquire pairs, β, β_1, β_2 contain no writes to X or acquire reads, and γ contains no reads from X or release-acquire pairs.

[7] In fact, it was carried out by a student with minimal understanding of weak memory consistency!

Domain:	$D \in \mathsf{Loc} \to \{\circ(v), \bullet(v), \top\}$		
Ordering:	$\forall v.\, \circ(v) \sqsubseteq \bullet(v) \sqsubseteq \top$		
Transitions:	$T(X)(X^{\texttt{na}} := v, \_)$	$= \circ(v)$	
	$T(X)(Y^{\texttt{rel}} := \_, \circ(v))$	$= \bullet(v)$	
	$T(X)(\_ := Y^{\texttt{acq}}, \bullet(v))$	$= \top$	
	$T(X)(\_, t)$	$= t$	otherwise

Figure 3.3: Store-to-load forwarding analysis

```
$\{X \mapsto \top\}$
$X^{\texttt{na}} := 42\,;$
$\{X \mapsto \circ(42)\}$
$l := Y^{\texttt{acq}}\,;$
**if** $l = 0$ **then**
    $\{X \mapsto \circ(42)\}$
    $a := X^{\texttt{na}}\,; \quad \rightsquigarrow \quad a := 42\,;$
    $\{X \mapsto \circ(42)\}$
    $Y^{\texttt{rel}} := 1$
    $\{X \mapsto \bullet(42)\}$
$\{X \mapsto \bullet(42)\}$
$b := X^{\texttt{na}} \quad \rightsquigarrow \quad b := 42$
```

Two loads from X are optimized to register assignments. To illustrate the analysis, the code is annotated with abstract tokens to X. The first instruction $X^{\texttt{na}} := 42$ induces UB if there is no permission on X. Therefore, the permission on X is guaranteed, with the memory value 42 at X (which is represented by $X \mapsto \circ(42)$). Since X is already permissioned, its value is not updated by the $l := Y^{\texttt{acq}}$, thereby maintaining the abstract state of X. Upon a conditional, we keep analyzing each branch separately, and then join the results. On the **then** branch, $a := X^{\texttt{na}}$ will load 42 from the memory as the abstract state indicates. For the next instruction, $Y^{\texttt{rel}} := 1$, the abstract state of X transitions to $X \mapsto \bullet(42)$ as the permission on X can be dropped by the release write, while the memory value at X is maintained. Finally, the branch is merged and the analysis results are joined (following the partial order on the abstract tokens). The effect of the last instruction, $b := X^{\texttt{na}}$, depends on the permission on X. If there is no permission on X, `undef` is read, which can be replaced by 42 by definition. In turn, the abstract state of X tells us 42 must be loaded if there is a permission on X. From the above analysis, we conclude that the two loads can be replaced with register assignments.

Figure 3.4: An example optimization by SLF including the underlying analysis in SEQ

Next, we focus on the SLF pass and describe the analysis and optimization in detail. The other passes are described in §3.5.1. Figure 3.3 depicts the analysis performed in the SLF pass, which forwards values written by stores to later loads, possibly across atomic operations, but not across a release-acquire pair. At every program point, the analysis assigns two kinds of information to each shared variable: a memory value to forward and a flag for detecting a release-acquire pair after the most recent write. This information is represented by the following abstract tokens:

- $X \mapsto \circ(v)$ indicates that v was written to X by the most recent write to X and no release write has been executed after the write;
- $X \mapsto \bullet(v)$ indicates that v was written to X by the most recent write to X and a release operation has been executed while a release-acquire pair has not; and
- $X \mapsto \top$ indicates any other case, in particular, the case when a release-acquire pair has been executed since the last write to X.

The analysis starts with the initial abstract state assigning $\top$ to every location in the initial program point. Then, it updates line-by-line the abstract state following the transition function T, which gets the current instruction and the token to a location X and returns the next abstract token to X. Roughly speaking, following the transformers in Fig. 3.3, the abstract state of X transitions to $\circ(v)$ for a non-atomic write $X^{\mathtt{na}} := v$; $\circ(v)$ transitions to $\bullet(v)$ for a release write; and $\bullet(v)$ transitions to $\top$ for an acquire read. To show termination, we have proved that the analysis reaches a fixpoint in at most three iterations when analyzing a loop.

Given the analysis result at each program point, SLF transforms a read $a := X^{\mathtt{na}}$ into a register assignment $a := v$ if the token to X is $\bullet(v)$ or $\circ(v)$ at that program point. Intuitively, having $X \mapsto \bullet(v)$ or $X \mapsto \circ(v)$ means that no release-acquire pair has been executed since v was written to X, thus the memory value of X is still v even if the thread has lost the permission to X. The transformation is sound since the thread will read v or `undef` from X depending on whether the permission to X

has been lost or not. Formally, a reachable SEQ state $\langle \sigma, P, F, M \rangle$ is related to the analysis result at the relevant program point as follows:

$$\forall X. \begin{cases} X \in P \wedge v \sqsubseteq M(X) & \text{if } X \mapsto \circ(v) \\ X \in P \Rightarrow v \sqsubseteq M(X) & \text{if } X \mapsto \bullet(v) \end{cases}$$

Figure 3.4 describes how the analysis and optimization work for a concrete program example.

The verification of the passes is executed by (*i*) establish the soundness of each analysis; and (*ii*) from the soundness, derive a simulation in SEQ between the source and target codes. The simulation relation in SEQ (given in §3.5.2) ensures advanced behavioral refinement as defined in §3.3.[8] This verification strategy follows the standard approach of CompCert, and, importantly, the optimizer is fully verified in Coq relying solely on sequential reasoning.

3.5 A Certified Optimizer in Detail

3.5.1 Optimizations in Detail

In this subsection, we describe analyses and transformations performed in the optimization passes, Load-to-Load Forwarding (LLF), Dead Store Elimination (DSE), and Loop Invariant Code Motion (LICM) in detail. The analyses for LLF and DSE are given in Fig. 3.5a and Fig. 3.5b respectively.

Load-to-Load Forwarding (LLF) forwards values read by loads to later loads, possibly across some atomic operations, but not across acquire accesses. The analysis assigns an abstract state, a location-wise set of registers, at every program point. Here, $X \mapsto R$ for some $R \subseteq \mathsf{Reg}$ indicates that the registers in R contain values loaded from X since the last acquire access. Note that an acuiqre access may invalidate such

[8]In fact, this is the same simulation used in the adequacy proof (see §3.7), so the optimizer correctness argument is independent of behavioral refinement in SEQ—it goes directly from simulation in SEQ to simulation in PS^{na}.

$$\begin{array}{lll} \text{Domain:} & D \in \mathsf{Loc} \to \mathcal{P}(\mathsf{Reg}) & \\ \text{Ordering:} & D_1 \sqsubseteq D_2 \triangleq \forall X.\ D_1(X) \supseteq D_2(X) & \\ \text{Transitions:} & T(X)(X^{\mathtt{na}} := v, t) & = \emptyset \\ & T(X)(a := X^{\mathtt{na}}, t) & = t \cup \{a\} \\ & T(X)(\_ := Y^{\mathtt{acq}}, t) & = \emptyset \\ & T(X)(\_, t) & = t \quad \text{otherwise} \end{array}$$

(a) Analysis for Load-to-Load Forwarding

$$\begin{array}{lll} \text{Domain:} & D \in \mathsf{Loc} \to \{ \circ, \bullet, \top \} & \\ \text{Ordering:} & \circ \sqsubseteq \bullet \sqsubseteq \top & \\ \text{Transitions:} & T_B(X)(X^{\mathtt{na}} := \_, t) & = \circ \\ & T_B(X)(\_ := X^{\mathtt{na}}, t) & = \top \\ & T_B(X)(Y^{\mathtt{rel}} := \_, \bullet) & = \top \\ & T_B(X)(\_ := Y^{\mathtt{acq}}, \circ) & = \bullet \\ & T_B(X)(\_, t) & = t \quad \text{otherwise} \end{array}$$

(b) Analysis for Dead Store Elimination

Figure 3.5: Analyses for optimizer passes

information by acquiring new values for the memory from the context. The analysis starts from the initial abstract state assigning $\emptyset$ to every location in the initial program point, and updates the abstract state following the transition function T. In particular, the analysis adds a register a to the abstract state of X when it meets a (non-atomic) read $a := X^{\mathtt{na}}$; and it empties the abstract state of any location when it meets an acquire access. Given the analysis result at each program point, LLF transforms a read $a := X^{\mathtt{na}}$ into a register assignment $a := b$ if there is a register b in the register set of X at that program point. Formally, any reachable SEQ state $\langle \sigma, P, F, M \rangle$ is related to the analysis result at that program point as follows (where $\sigma.\mathtt{rs}$ indicates a register file assigning a value to each register):

$$\forall X\ r.\ X \in P \land r \in R \Rightarrow \sigma.\mathtt{rs}(r) \sqsubseteq M(X) \quad \text{for } X \mapsto R$$

Dead Store Elimination (DSE) removes dead stores which are overwritten by other stores, possibly across some atomic operations, but not across release-acquire pairs. The analysis of DSE is interesting in that unlike SLF or LLF, it analyzes given code backward because it has to analyze if a location will be overwritten in the future or not. Specifically, at every program point, the analysis assigns to each share variable a flag indicating if there is a later store before facing a release-acquire pair or a read from the corresponding location. This information is represented by the following abstract tokens:

- $X \mapsto \circ$ indicates that there is a overwriting store in the future and no acquire read or a read from X can be executed in the middle;
- $X \mapsto \bullet$ indicates that there is a overwriting store in the future and an acquire read may be executed in the middle while a release write or a read from X may not; and
- $X \mapsto \top$ indicates any other case, in particular, including the case when there is a overwriting store in the future, but a release-acquire pair or a read from X can be executed in the middle.

The *backward* transition function T_B, which gets the current instruction and the token to a location X and returns the abstract token to X *before* the instruction, is given in Fig. 3.5b. Roughly speaking, the abstract state of X transitions to $\circ$ for a non-atomic write to X; $\circ$ transitions to $\bullet$ for an acquire read; $\bullet$ transitions to $\top$ for a release write; and any token transitions to $\top$ for a read from X. Given the analysis result at each program point, DSE transformas a write $X^{\mathbf{na}} := \_$ into a **skip** if the token to X is $\circ$ or $\bullet$ at that program point. Formally, any reachable SEQ state $S = \langle \sigma, P, F, M \rangle$ is related to the analysis result at that program point as follows: for any X, (*i*) in any execution of S under SEQ, X is overwritten before executing a read from X or an acquire read if $X \mapsto \circ$; and (*ii*) in any execution of S under SEQ, X is overwritten before executing a read from X or a release-acquire pair if $X \mapsto \bullet$.

Loop Invariant Code Motion (LICM) is implemented in two stages: (*i*) introducing irrelevant loads; and (*ii*) forwarding the loaded values of the introduced loads to the loads inside the loop using the LLF pass we discussed above. Since introducing an irrelevant load is unconditionally sound in SEQ (*i.e.*, no analysis is required), it is

$$
\begin{aligned}
&\langle \sigma_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \overset{A}{\sim}_{P,R} \langle \sigma_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle \triangleq \\
&((\sigma_{\mathrm{tgt}} \neq \bot) \ \wedge \\
&\ (\forall\, v_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} = \mathbf{return}(v_{\mathrm{tgt}})) \Rightarrow \\
&\qquad \exists\, v_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} = \mathbf{return}(v_{\mathrm{src}}))\ \wedge\ ((v_{\mathrm{src}}, v_{\mathrm{tgt}}) \in A)\ \wedge\ (F_{\mathrm{tgt}} \cup R \sqsubseteq F_{\mathrm{src}})\ \wedge\ (M_{\mathrm{tgt}} \sqsubseteq M_{\mathrm{src}}))\ \wedge \\
&\ (\forall\, \sigma'_{\mathrm{tgt}}\ F'_{\mathrm{tgt}}\ M'_{\mathrm{tgt}}.\ (\langle \sigma_{\mathrm{tgt}}, P, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle \to \langle \sigma_{\mathrm{tgt}}, P, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle) \Rightarrow \\
&\qquad \exists\, \sigma'_{\mathrm{src}}\ F'_{\mathrm{src}}\ M'_{\mathrm{src}}.(\langle \sigma_{\mathrm{src}}, P, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \to^{*} \langle \sigma'_{\mathrm{src}}, P, F'_{\mathrm{src}}, M'_{\mathrm{src}}\rangle)\ \wedge\ (\langle \sigma'_{\mathrm{src}}, F'_{\mathrm{src}}, M'_{\mathrm{src}}\rangle \overset{A}{\sim}_{P,R} \langle \sigma'_{\mathrm{tgt}}, F'_{\mathrm{tgt}}, M'_{\mathrm{tgt}}\rangle))\ \wedge \\
&\ (\forall\, v\ \sigma'_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} \xrightarrow{\mathtt{choose}(v)} \sigma'_{\mathrm{tgt}}) \Rightarrow \\
&\qquad \exists\, \sigma'_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} \xrightarrow{\mathtt{choose}(v)} \sigma'_{\mathrm{src}})\ \wedge\ (\langle \sigma'_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \overset{A}{\sim}_{P,R} \langle \sigma'_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle))\ \wedge \\
&\ (\forall\, e\ P'\ \sigma'_{\mathrm{tgt}}\ F'_{\mathrm{tgt}}\ M'_{\mathrm{tgt}}.\ \ (\forall\, X\ v\ \sigma'_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} \xrightarrow{\mathtt{R}^{\mathtt{rlx}}(X,v)} \sigma'_{\mathrm{tgt}}) \Rightarrow \\
&\qquad \exists\, \sigma'_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} \xrightarrow{\mathtt{R}^{\mathtt{rlx}}(X,v)} \sigma'_{\mathrm{src}})\ \wedge\ (\langle \sigma'_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \overset{A}{\sim}_{P,R} \langle \sigma'_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle))\ \wedge \\
&\ (\forall\, X\ v_{\mathrm{tgt}}\ \sigma'_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} \xrightarrow{\mathtt{W}^{\mathtt{rlx}}(X,v)} \sigma'_{\mathrm{tgt}}) \Rightarrow \\
&\qquad \exists\, \sigma'_{\mathrm{src}}\ v_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} \xrightarrow{\mathtt{W}^{\mathtt{rlx}}(X,v)} \sigma'_{\mathrm{src}})\ \wedge\ (v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}})\ \wedge\ (\langle \sigma'_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \overset{A}{\sim}_{P,R} \langle \sigma'_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}}\rangle))\ \wedge \\
&\ (\forall\, X\ v\ \sigma'_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} \xrightarrow{\mathtt{R}^{\mathtt{acq}}(X,v)} \sigma'_{\mathrm{tgt}}) \Rightarrow \\
&\qquad \exists\, \sigma'_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} \xrightarrow{\mathtt{R}^{\mathtt{acq}}(X,v)} \sigma'_{\mathrm{src}})\ \wedge\ (F_{\mathrm{tgt}} \cup R \sqsubseteq F_{\mathrm{src}})\ \wedge\ \forall\, P'\ V.\ ((P \subseteq P')\ \wedge\ (dom(V) = P' \setminus P))\ \Rightarrow \\
&\qquad\quad (\langle \sigma'_{\mathrm{src}}, F_{\mathrm{src}}, (\lambda X.(X \in dom(V))?V(X) : M_{\mathrm{src}}(X))\rangle \overset{A}{\sim}_{P',\emptyset} \langle \sigma'_{\mathrm{tgt}}, F_{\mathrm{tgt}}, (\lambda X.(X \in dom(V))?V(X) : M_{\mathrm{tgt}}(X))\rangle))\ \wedge \\
&\ (\forall\, X\ v_{\mathrm{tgt}}\ \sigma'_{\mathrm{tgt}}.\ (\sigma_{\mathrm{tgt}} \xrightarrow{\mathtt{W}^{\mathtt{rel}}(X,v_{\mathrm{tgt}})} \sigma'_{\mathrm{tgt}}) \Rightarrow \\
&\qquad \exists\, v_{\mathrm{src}}\ \sigma'_{\mathrm{src}}.\ (\sigma_{\mathrm{src}} \xrightarrow{\mathtt{W}^{\mathtt{rel}}(X,v_{\mathrm{src}})} \sigma'_{\mathrm{src}})\ \wedge\ (v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}})\ \wedge\ \forall\, P'.\ (P \subseteq P')\ \Rightarrow \\
&\qquad\quad \exists R'.\ (F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}} \cup R')\ \wedge\ (\forall X \notin R'.M_{\mathrm{tgt}}(X) \sqsubseteq M_{\mathrm{src}}(X))\ \wedge\ (\langle \sigma'_{\mathrm{src}}, \emptyset, M_{\mathrm{src}}\rangle \overset{A}{\sim}_{P',R'} \langle \sigma'_{\mathrm{tgt}}, \emptyset, M_{\mathrm{tgt}}\rangle))\ \wedge \\
&\ (\forall\, \Omega.\ \exists\, P'\ \sigma'_{\mathrm{src}}\ F'_{\mathrm{src}}\ M'_{\mathrm{src}}\ tr. \\
&\qquad (\langle \sigma_{\mathrm{src}}, P, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \xrightarrow{tr} \langle \sigma'_{\mathrm{src}}, P', F'_{\mathrm{src}}, M'_{\mathrm{src}}\rangle)\ \wedge\ (tr \in \mathsf{Tr}(\Omega))\ \wedge\ (\mathtt{R}^{\mathtt{acq}}(_) \notin tr)\ \wedge \\
&\qquad ((\sigma'_{\mathrm{src}} = \bot)\ \vee\ (F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}} \cup \bigcup\{F \mid \mathtt{W}^{\mathtt{rel}}(_,_,_,_,F,_) \in tr\})))\ \vee \\
&(\forall\, \Omega.\ \exists\, P'\ F'_{\mathrm{src}}\ M'_{\mathrm{src}}\ tr. \\
&\qquad (\langle \sigma_{\mathrm{src}}, P, F_{\mathrm{src}}, M_{\mathrm{src}}\rangle \xrightarrow{tr} \langle \bot, P', F'_{\mathrm{src}}, M'_{\mathrm{src}}\rangle)\ \wedge\ (tr \in \mathsf{Tr}(\Omega))\ \wedge\ (\mathtt{R}^{\mathtt{acq}}(_) \notin tr))
\end{aligned}
$$

$$\sigma_{\mathrm{src}} \sim_A \sigma_{\mathrm{tgt}} \triangleq \forall\, M\ F\ P.\ \langle \sigma_{\mathrm{src}}, F, M\rangle \overset{A}{\sim}_{P,\emptyset} \langle \sigma_{\mathrm{tgt}}, F, M\rangle$$

Figure 3.6: A simulation relation for SEQ

enough for LICM pass to decide which load needs to be introduced. Indeed, LICM analyzes each loop body and collect the shared variables that can be potentially hoisted. Note that this analysis only affects the performance of the optimized code, but not the correctness of the optimization pass itself. Once the loads are introduced before each loop, running LLF pass transforms the loads inside the loop into register assignments, resulting in the code where loop invariant loads are hoisted.

3.5.2 Simulation Relation in SEQ

We define a simulation relation to establish the soundness of the optimization passes following previous work [53, 66]. The simulation relation in SEQ is given in Fig. 3.6. We also provide the structural rules, given in Fig. 3.7, which allow reasoning about

$$\text{(REFLEXIVITY)}\ \frac{A \text{ is reflexive}}{\sigma \sim_A \sigma} \qquad \text{(MONOTONICITY)}\ \frac{A \subseteq A' \quad \sigma_{\text{src}} \sim_A \sigma_{\text{tgt}}}{\sigma_{\text{src}} \sim_{A'} \sigma_{\text{tgt}}} \qquad \text{(RETURN)}\ \frac{(v_{\text{src}}, v_{\text{tgt}}) \in A}{\mathbf{return}(v_{\text{src}}) \sim_A \mathbf{return}(v_{\text{tgt}})}$$

$$\text{(BIND)}\ \frac{\sigma_{\text{src}} \sim_A \sigma_{\text{tgt}} \qquad \forall (v_{\text{src}}, v_{\text{tgt}}) \in A0.\ k_{\text{src}}(v_{\text{src}}) \sim_{A1} k_{\text{tgt}}(v_{\text{tgt}})}{(\sigma_{\text{src}} >\!\!>\!= k_{\text{src}}) \sim_{A1} (\sigma_{\text{tgt}} >\!\!>\!= k_{\text{tgt}})} \qquad \text{(ITERATION)}\ \frac{\forall (v_{\text{src}}, v_{\text{tgt}}) \in A0.\ k_{\text{src}}(v_{\text{src}}) \sim_{A0+A1} k_{\text{tgt}}(v_{\text{tgt}})}{\texttt{iter}(k_{\text{src}})(i) \sim_{A1} \texttt{iter}(k_{\text{tgt}})(i)}$$

Figure 3.7: Compatibility Lemmas

a part of a program and composing it with larger contexts. Note that the bind and iteration operators are of interaction trees, which we use to define WHILE language. As stated in Thm. 8, the simulation relation in SEQ implies behavioral refinement in SEQ,

Theorem 8. If $\sigma_{\text{src}} \sim_A \sigma_{\text{tgt}}$ for a relation A, then $\sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$.

For the optimization passes, we show that the optimized program can be simulated by the original one (Lemma 9), and conclude the soundness (Thm. 10) of the optimizations by Thm. 8.

Lemma 9. For $f \in \{SLF, LLF, DSE, LICM\}$, $\sigma \sim_A f(\sigma)$.

Theorem 10. For $f \in \{SLF, LLF, DSE, LICM\}$, $f(\sigma) \sqsubseteq_{\text{W}} \sigma$.

3.6 Non-atomics in the Promising Semantics

We present the extension of PS2.1 with non-atomic accesses, which we denote by $\text{PS}^{\textsf{na}}$. At the core of this extension is an operational race detection, so UB is invoked on write-write races and `undef` is read on read-write races. Unlike in SEQ (§3.2), we allow the mixing of atomic and non-atomic accesses to the same location (so we assume one set $\textsf{Loc}$ of locations), which means that a race may involve only one non-atomic access. As for SEQ, we only present a simplified fragment of the full model omitting fences, RMWs, release sequences, reservations/cancellations and system calls, which are all covered by our Coq formalization.

Figure 3.8 presents the thread configuration steps and the machine steps. Next, we discuss the new parts, highlighted in the figure. We refer the reader to §1.2 for explanations of the atomics fragment of $\mathrm{PS}^{\mathbf{na}}$, which is identical to PS2.1. We add steps for normal (successful) non-atomic accesses and for racy (both atomic and non-atomic) accesses.

Normal non-atomic accesses are handled by READ and WRITE transitions. A non-atomic read (READ) from X behaves exactly as a relaxed one: reads from a message with timestamp t that is greater than or equal to the thread's view of X, and updates the view to include t. In turn, non-atomic writes (WRITE) require a non-trivial extension. When a thread executes a non-atomic write to a location X, it may add multiple arbitrary messages with the bottom view (denoted by $\bot$, a view smaller than any other view) to X before adding a message with the appropriate value (MEMORY: NA-WRITE). In addition, some of the messages preceding the final one may be valueless *non-atomic messages* of the form $u = X@t \in \mathsf{NAMsg}$, which we introduce for detecting races.[9] Writing multiple messages in one non-atomic write allows the splitting of non-atomic writes which is needed in order to allow certain program transformations (see Example 26).

Example 26. We present an example that justifies allowing non-atomic writes to add multiple arbitrary messages to the memory. Consider the following program where π_2 is optimized as shown:

$$
\begin{array}{l}
a := X^{\mathtt{na}}\,; \\
Y^{\mathtt{rlx}} := a
\end{array}
\left\|
\begin{array}{l}
b := Y^{\mathtt{rlx}}\,; \\
c := \mathbf{freeze}(b)\,; \\
\mathbf{if}\ c = 1\ \mathbf{then} \\
\quad X^{\mathtt{na}} := 1\,; \\
\quad \mathbf{print}(1) \\
\mathbf{else} \\
\quad X^{\mathtt{na}} := 2
\end{array}
\right.
\quad\leadsto\quad
\begin{array}{l}
b := Y^{\mathtt{rlx}}\,; \\
c := \mathbf{freeze}(b)\,; \\
X^{\mathtt{na}} := 2\,; \\
\mathbf{if}\ c = 1\ \mathbf{then} \\
\quad X^{\mathtt{na}} := 1\,; \\
\quad \mathbf{print}(1) \quad /\!/\ \textit{reachable!}
\end{array}
$$

Suppose that a non-atomic write is only allowed write a single message as a relaxed write does. Then, after the optimization, π_2 is allowed to print 1 by entering the

[9] We assume that $u.\mathtt{view} = \bot$ for $u \in \mathsf{NAMsg}$.

$v \in \mathsf{Val}$	value	$t \in \mathsf{Time} \triangleq \{0\} \cup \mathbb{Q}^+$	timestamp	σ	thread-local program state
$X, Y, Z \in \mathsf{Loc}$	location	$V \in \mathsf{View} \triangleq (\mathsf{Loc} \to \mathsf{Time}) \cup \{\bot\}$	view	$T = \langle \sigma, V, P \rangle \in \mathsf{Lts}$	thread state
$o_{\mathtt{R}} \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}\}$	read access mode	$m = \langle X@t, v, V \rangle \in \mathsf{Msg}$	message	$\langle T, M \rangle$	thread configuration
$o_{\mathtt{W}} \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}\}$	write access mode	$u = X@t \in \mathsf{NAMsg}$	non-atomic message	$\mathcal{T} : \mathsf{Tid} \to \mathsf{Lts}$	thread state mapping
$\pi \in \mathsf{Tid} \triangleq \{\pi_1, \pi_2, \ldots\}$	thread identifier	$M, P \subseteq \mathsf{Msg} \cup \mathsf{NAMsg}$	memory/promise set	$\langle \mathcal{T}, M \rangle$	machine state

(MEMORY: NEW)

$$\frac{}{\langle P, M \rangle \xrightarrow{m} \langle P, M \uplus \{m\} \rangle}$$

(MEMORY: FULFILL)

$$\frac{m \in P}{\langle P, M \rangle \xrightarrow{m} \langle P \setminus \{m\}, M \rangle}$$

(PROMISE)

$$\frac{m \in \mathsf{Msg} \cup \mathsf{NAMsg}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \sigma, V, P \uplus \{m\} \rangle, M \uplus \{m\} \rangle}$$

(LOWER)

$$\frac{\begin{array}{c} m = \langle X@t, v, V_{\mathtt{m}} \rangle \in P \\ m' = \langle X@t, v', V'_{\mathtt{m}} \rangle \qquad v \sqsubseteq v' \qquad V'_{\mathtt{m}} \sqsubseteq V_{\mathtt{m}} \\ P' = P \setminus \{ m \} \cup \{ m' \} \qquad M' = M \setminus \{ m \} \cup \{ m' \} \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \sigma, V, P' \rangle, M' \rangle}$$

(MEMORY: NA-WRITE)

$$\frac{\begin{array}{c} \langle P, M \rangle \xrightarrow{m_1} \ldots \xrightarrow{m_n} \langle P_n, M_n \rangle \xrightarrow{m} \langle P', M' \rangle \qquad n \geq 0 \\ m_1, \ldots, m_n \in \mathsf{Msg} \cup \mathsf{NAMsg} \\ m_1.\mathtt{loc} = \ldots = m_n.\mathtt{loc} = m.\mathtt{loc} \qquad t < m_1.\mathtt{t}, \ldots, m_n.\mathtt{t} < m.\mathtt{t} \\ m_1.\mathtt{view} = \ldots = m_n.\mathtt{view} = m.\mathtt{view} = \bot \end{array}}{\langle P, M \rangle \xrightarrow{t,m}_{\mathtt{na}} \langle P', M' \rangle}$$

(WRITE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathtt{W}^{o_{\mathtt{W}}}(X,v)} \sigma' \qquad m = \langle X@t, v, V_{\mathtt{m}} \rangle \qquad V(X) < t \qquad V' = V[X \mapsto t] \\ o_{\mathtt{W}} = \mathtt{na} \;\Rightarrow\; V_{\mathtt{m}} = \bot \;\wedge\; \langle P, M \rangle \xrightarrow{V(X),m}_{\mathtt{na}} \langle P', M' \rangle \\ o_{\mathtt{W}} = \mathtt{rlx} \;\Rightarrow\; V_{\mathtt{m}} = [X \mapsto t] \;\wedge\; \langle P, M \rangle \xrightarrow{m} \langle P', M' \rangle \\ o_{\mathtt{W}} = \mathtt{rel} \;\Rightarrow\; V_{\mathtt{m}} = V' \;\wedge\; \langle P, M \rangle \xrightarrow{m} \langle P', M' \rangle \;\wedge\; \forall m \in P|^{\mathsf{Msg}}_X.\ m.\mathtt{view} = \bot \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \sigma', V', P' \rangle, M' \rangle}$$

(READ)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathtt{R}^{o_{\mathtt{R}}}(X,v)} \sigma' \qquad m = \langle X@t, v, V_{\mathtt{m}} \rangle \in M \qquad V(X) \leq t \\ o_{\mathtt{R}} \neq \mathtt{acq} \;\Rightarrow\; V' = V \sqcup [X \mapsto t] \\ o_{\mathtt{R}} = \mathtt{acq} \;\Rightarrow\; V' = V \sqcup [X \mapsto t] \sqcup V_{\mathtt{m}} \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \sigma', V', P \rangle, M \rangle}$$

(RACE-HELPER)

$$\frac{\begin{array}{c} m \in M \setminus P \\ m.\mathtt{loc} = X \qquad V(X) < m.\mathtt{t} \\ o \neq \mathtt{na} \Rightarrow m \in \mathsf{NAMsg} \end{array}}{\langle V, P, M \rangle \text{ is racy on } X \text{ with } o}$$

(RACY-READ)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathtt{R}^{o_{\mathtt{R}}}(X,\mathtt{undef})} \sigma' \\ \langle V, P, M \rangle \text{ is racy on } X \text{ with } o_{\mathtt{R}} \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \sigma', V, P \rangle, M \rangle}$$

(RACY-WRITE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathtt{W}^{o_{\mathtt{W}}}(X,\_)} \sigma' \\ \langle V, P, M \rangle \text{ is racy on } X \text{ with } o_{\mathtt{W}} \\ \forall m \in P.\ V(m.\mathtt{loc}) < m.\mathtt{t} \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \bot, V, \emptyset \rangle, M \rangle}$$

(SILENT)

$$\frac{\sigma \to \sigma' \qquad \sigma' \neq \bot}{\begin{array}{c} \langle \langle \sigma, V, P \rangle, M \rangle \to \\ \langle \langle \sigma', V, P \rangle, M \rangle \end{array}}$$

(CHOOSE)

$$\frac{\sigma \xrightarrow{\mathtt{choose}(v)} \sigma'}{\begin{array}{c} \langle \langle \sigma, V, P \rangle, M \rangle \to \\ \langle \langle \sigma', V, P \rangle, M \rangle \end{array}}$$

(FAIL)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathtt{fail}} \bot \\ \forall m \in P.\ V(m.\mathtt{loc}) < m.\mathtt{t} \end{array}}{\langle \langle \sigma, V, P \rangle, M \rangle \to \langle \langle \bot, V, \emptyset \rangle, M \rangle}$$

(MACHINE: NORMAL)

$$\frac{\begin{array}{c} \langle \mathcal{T}(\pi), M \rangle \to^+ \langle T', M' \rangle \\ \langle T', M' \rangle \to^* \langle \langle \sigma'', V'', \emptyset \rangle, M'' \rangle \end{array}}{\langle \mathcal{T}, M \rangle \to \langle \mathcal{T}[\pi \mapsto T'], M' \rangle}$$

(MACHINE: FAILURE)

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

Figure 3.8: Transitions of PS$^{\mathtt{na}}$ (differences w.r.t. the corresponding fragment of PS2.1 are highlighted)

if-branch through following execution:

(i) π_2 promises $X = 2$;

(ii) π_1 reads **`undef`** from X, writes it back to Y;

(iii) π_2 reads **`undef`** from Y, freezes[10] the read value (*i.e.*, **`undef`**) to 1, and prints 1 by executing the rest of the thread's code.

However, π_2 before the optimization cannot print 1 unless a non-atomic write is allowed to add multiple messages to the memory. Indeed, once π_2 promises $X = 2$, it cannot enter the if-branch since the promise $X = 2$ cannot be fulfilled through the write $X^{\mathtt{na}} := 1$. (Note that if a non-atomic write can write multiple messages, including $X = 2$ in this example, the promise $X = 2$ can be fulfilled through the write $X^{\mathtt{na}} := 1$.) In addition, π_2 cannot promise $X = 1$ because it cannot be certified. Therefore, there is no execution where π_2 prints 1, which makes this optimization unsound.

[10]We place **freeze** here to prevent π_2 from invoking UB due to the branching on **`undef`**. Note that **freeze** returns the given value when a normal (non-**`undef`**) value is passed and returns an arbitrary normal value when **`undef`** is passed.

Racy accesses are naturally defined: a non-atomic access to X is racy if the thread is unaware of some message with location X ($V(X) < m.\mathtt{t}$), and an atomic access to X is racy if the thread is unaware of some *non-atomic* message with location X. Using RACE-HELPER, a thread reads `undef` when performing a racy read (RACY-READ), and invokes UB on a racy write (RACY-WRITE).

For supporting the compiler transformation that replaces an `undef` by a non-`undef` value, we note that the promise lowering step in PS$^{\mathtt{na}}$ (LOWER), which allows threads to modify their own promises, also allows to change a non-`undef` value of a promise to `undef`. This lower operation is introduced to explain a common compiler transformation that replaces an `undef` in the source program with a non-`undef` value (see Example 27).

Example 27. PS$^{\mathtt{na}}$ allows a thread to *lower* its outstanding promises by changing the message value from v_1 with v_2 where $v_1 \sqsubseteq v_2$. To see why the lower operation is required, consider the following transformation:

$$
\begin{array}{l}
c := Y^{\mathtt{rlx}}\,; \\
\textbf{if } c = 1 \textbf{ then} \\
\quad X^{\mathtt{rlx}} := 1 \\
\textbf{else} \\
\quad X^{\mathtt{rlx}} := \mathtt{undef}
\end{array}
\;\rightsquigarrow\;
\begin{array}{l}
c := Y^{\mathtt{rlx}}\,; \\
\textbf{if } c = 1 \textbf{ then} \\
\quad X^{\mathtt{rlx}} := 1 \\
\textbf{else} \\
\quad X^{\mathtt{rlx}} := \mathtt{undef}
\end{array}
\;\rightsquigarrow\;
\begin{array}{l}
c := Y^{\mathtt{rlx}}\,; \\
\textbf{if } c = 1 \textbf{ then} \\
\quad X^{\mathtt{rlx}} := 1 \\
\textbf{else} \\
\quad X^{\mathtt{rlx}} := 1
\end{array}
\;\rightsquigarrow\;
\begin{array}{l}
X^{\mathtt{rlx}} := 1\,; \\
c := Y^{\mathtt{rlx}}
\end{array}
$$

After the second transformation, which is marked in red, the thread can promise $X = 1$ before executing $c := Y^{\mathtt{rlx}}$ and later fulfill the promise by taking else-branch. In contrast, without the lower operation, the thread before the optimization cannot fulfill its promise $X = 1$ by taking else-branch since the write $X^{\mathtt{rlx}} := \mathtt{undef}$ cannot fulfill the promise $X = 1$. The lower operation solves this problem by allowing the thread to first lower its promise $X = 1$ to $X = \mathtt{undef}$ and then fulfill the promise through the write $X^{\mathtt{rlx}} := \mathtt{undef}$.

Machine steps A machine state, which consists of the different thread states ($\mathcal{T}$) and a main memory (M), can take a step by one of the threads taking a sequence of steps (MACHINE: NORMAL), possibly invoking UB (MACHINE: FAILURE). Normal steps (MACHINE: NORMAL) require "certification": the thread that passes control to the scheduler has to show that by running alone it can fulfill all its promises.

Example 28. The following demonstrates how promises and the racy read step work:

$$\left.\begin{array}{l} a := X^{\texttt{na}}\,; \quad /\!/\texttt{undef} \\ Y^{\texttt{rlx}} := 1 \end{array}\right\| \begin{array}{l} b := Y^{\texttt{rlx}}\,; \\ \textbf{if } b = 1 \textbf{ then} \\ \quad X^{\texttt{na}} := 1 \end{array}$$

Here, the left thread may promise $Y = 1$, since by running alone, it is able to execute the read from X and fulfill its promise. Then, the right thread reads 1 from Y and writes 1 to X. (Other messages may be also added to X before the $X = 1$ message.) Now, the non-atomic read from X of the left thread is racy since there is a message of X with timestamp larger than the thread's view of X. Thus, the thread reads `undef` from X and fulfills the promise $Y = 1$.

Results We ported to PS$^{\texttt{na}}$ the soundness proofs of all thread-local transformations and data-race-freedom guarantees for PS2.1. In addition, we proved that strengthening non-atomic accesses to atomic accesses is sound. Since relaxed accesses and non-atomics are both compiled to plain machine accesses, the soundness of mapping schemes to hardware follows from the soundness of this strengthening and of the mapping PS2.1 to hardware as shown in [4, 1].

Behavioral refinement A behavior in PS$^{\texttt{na}}$ is defined as in Def. 2. We write $r_{\text{tgt}} \sqsubseteq r_{\text{src}}$ if either $r_{\text{src}} = \bot$ or $\forall \pi.\, r_{\text{tgt}}(\pi) \sqsubseteq r_{\text{src}}(\pi)$. Behavioral refinement in PS$^{\texttt{na}}$ is defined as follows.

Definition 20. A concurrent program state $\sigma^1_{\text{tgt}} \| \dots \| \sigma^n_{\text{tgt}}$ *behaviorally refines* a state $\sigma^1_{\text{src}} \| \dots \| \sigma^n_{\text{src}}$, denoted by $\sigma^1_{\text{tgt}} \| \dots \| \sigma^n_{\text{tgt}} \sqsubseteq_{\text{PS}^{\texttt{na}}} \sigma^1_{\text{src}} \| \dots \| \sigma^n_{\text{src}}$, if whenever we have $\langle \lambda\pi.\, \langle \sigma^\pi_{\text{tgt}}, V_{\mathsf{init}}, \emptyset \rangle, M_{\mathsf{init}} \rangle \Downarrow r_{\text{tgt}}$, there exists r_{src} such that $r_{\text{tgt}} \sqsubseteq r_{\text{src}}$ and $\langle \lambda\pi.\, \langle \sigma^\pi_{\text{src}}, V_{\mathsf{init}}, \emptyset \rangle, M_{\mathsf{init}}$ r_{src}. (Here, V_{init} is the initial thread view assigning the timestamp 0 to every location; $\emptyset$ is the initial empty set of promises; and M_{init} is the initial memory consisting of an initialization message $\langle X@0, 0, \bot \rangle$ for every $X \in \mathsf{Loc}$.)

3.7 Adequacy of Sequential Reasoning

In this section, we state the adequacy of reasoning in SEQ w.r.t. PS^{na}, outline the main challenges in the proof, and discuss our approach to overcome them. First, we define deterministic programs, which is needed below.

Definition 21. A program state σ is *deterministic* if for every σ_0 reachable from σ (*i.e.*, $\sigma \rightarrow^* \sigma_0$), if both $\sigma_0 \xrightarrow{e_1} \sigma_1$ and $\sigma_0 \xrightarrow{e_2} \sigma_2$, then one of the following holds: (i) $e_1 = e_2$ and $\sigma_1 = \sigma_2$; (ii) $e_1 = \mathtt{R}^o(X, v_1)$, $e_2 = \mathtt{R}^o(X, v_2)$, and $v_1 \neq v_2$; or (iii) $e_1 = \mathtt{choose}(v_1)$, $e_2 = \mathtt{choose}(v_2)$, and $v_1 \neq v_2$.

Theorem 11 (Adequacy)**.** If $\sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$ (Def. 19) and σ_{src} is deterministic, then $\sigma_{\text{tgt}}||\sigma_1|| \dots ||\sigma_n \sqsubseteq_{\text{PS}^{\text{na}}} \sigma_{\text{src}}||\sigma_1|| \dots ||\sigma_n$ (Def. 20) for any programs $\sigma_1, \dots, \sigma_n$.

To prove this theorem, we first show that $\sigma_{\text{tgt}} \sqsubseteq_{\text{W}} \sigma_{\text{src}}$ implies the existence of a simulation relation between the source and target in SEQ (detailed in §3.5.2). Then, we show that a simulation in SEQ implies the existence of a simulation in PS^{na}. For this purpose, we lift steps in SEQ to thread steps in PS^{na}. This raises three significant challenges. First, there is a large gap between SEQ's simple states and the complex states of PS^{na}. Second, in PS^{na}, we should consider interference by other threads at every point, whereas in SEQ, memory states are changed only in release/acquire steps. Third, we need to show how promise steps of the target in PS^{na} are simulated by the source and establish a PS^{na} certification execution for every step of the source.

The key idea for the first point is that even though PS^{na} has complex states, not all its complexity affects non-atomic steps. In fact, a memory in SEQ can be seen as an approximation of a state in PS^{na} capturing only the part related to non-atomic steps. The value of a location X in SEQ correspond to the value of the message pointed by the thread view on X in PS^{na}, and a permission on X in SEQ means that there is no racy message with the thread in PS^{na}. Since non-atomic and relaxed accesses do not change the thread view on other locations, states in SEQ are not changed after non-atomic and relaxed accesses. In turn, an acquire read in PS^{na} may increase the thread view, which corresponds to the modified values and gained permissions in acquire steps of SEQ.

For the second point, we need a novel insight on the promising semantics: in a machine step, it suffices to have promise steps followed by non-promise steps ending with a release write (or thread termination). This implies that racy messages of other threads are added only when a release write is executed, which corresponds to SEQ losing permissions only on a release write.

For the third point, we construct the certification steps of the source execution from those of the target. The challenge here is the two cases where the target thread fulfills its promise while the source cannot: (*i*) when there is no source step corresponding to a write step of the target fulfilling a promise; and (*ii*) when the written message by the source has a different value than the target's. This challenge is addressed by the commitment set of the advanced refinement, which ensures that, in both cases, the source thread should be able to write to the problematic locations in the future, thereby allowing the source to establish its certification.

Mixing of atomics and non-atomics to the same location The above proof sketch does not work well under the presence of mixing of atomic and non-atomic accesses to the same location. This is why we assume there is no mixing of atomics and non-atomics to the same location.

In particular, $\text{PS}^{\textbf{na}}$ allows a value of existing message that is pointed by a thread's view to be lowered by another thread (see Example 27). However, this is not the case in SEQ: a memory value in SEQ is only updated by executing acquire accesses. This gap between SEQ and $\text{PS}^{\textbf{na}}$ invalidates the proof sketch above.

A possible solution to bridge this gap would be extending SEQ to allow memory values to be lowered to `undef` when release accesses are executed. Nevertheless, we chose to prohibit mixing of atomic and non-atomic accesses to the same location for two reasons: (*i*) to keep SEQ as simple as possible; and (*ii*) to make the proof of the adequacy theorem easier. By disallowing the mixing, one can easily show that a thread's view to a non-atomic location never points to an oustainding promise of another thread, thereby lifting the above problem of the message value being lowered by another thread.

Remark 10. Theorem 11 does not hold without the determinism premise. This stems from a drawback of the promising semantics (rather than due to SEQ) that we encountered while developing SEQ. Concretely, the promising semantics (PS as well as PS2 and PS2.1) disallows the reordering of an internal non-deterministic choice[11] followed by a release write. By exposing non-deterministic choices (via `choose(_)` labels), we invalidate these reorderings in SEQ and obtain adequacy for deterministic programs. (Nevertheless, the reordering of non-deterministic choices and non-atomic accesses is fully allowed by SEQ.) We believe that Thm. 11 holds for a properly fixed version of PS without explicit `choose(_)` events. We leave to future work to improve the promising semantics to allow this reordering, which will allow one to remove the `choose(_)` labels from SEQ.

The source of the problem is that release writes explicitly block promises with non-$\perp$ message view to the same location (*i.e.*, a thread transition for a release write to a location X requires that the thread has no promise with non-$\perp$ message view to X.) Consider the following program:

```
                    ||  b := freeze(undef) ;                      X^rel := 0 ;
                    ||  X^rel := 0 ;                              b := freeze(undef) ;
                    ||  if b = 1 then                             if b = 1 then
                    ||    c := Y^rlx ;                              c := Y^rlx ;
a := X^rlx ;        ||    if c = 1 then                      ⇝      if c = 1 then
Y^rlx := a          ||      X^rlx := 1 ;                              X^rlx := 1 ;
                    ||      print(1)  // not reachable!               print(1)  // reachable!
                    ||  else                                      else
                    ||    X^rlx := 1                                X^rlx := 1
```

Here, π_2 is optimized by reordering the **freeze** instruction with the release write to x. We observe that π_2 printing 1 is observable after the optimization while it is not before. Indeed, we note that π_2 can be further optimized so that printing 1 is observable even under a sequentially consistent execution.

First, π_2 can print 1 through following execution:

[11]a representative example of such non-determinism is **freeze** instruction of LLVM IR [58].

(i) π_2 writes 0 to X;

(ii) π_2 promises $X = 1$ (certifying it by freezing `undef` to 0);

(iii) π_1 reads 1 from X and writes 1 to Y;

(iv) π_2 freezes `undef` to 1 (unlike it did in the certification), enters the if-branch, reads 1 from Y, fulfills $X = 1$, and prints 1.

However, this behavior is not observable by π_2 before the optimization because the thread cannot promise $X = 1$ before freezing `undef` due to the release write to X that blocks the promise. Indeed, if π_2 freezes `undef` to 1, it cannot promise $X = 1$ since the promise cannot be certified (*i.e.*, π_2 cannot execute to write $X = 1$ in isolation.) Otherwise, it will not have any execution printing 1 as $b \neq 1$ is already determined to be false.

Therefore, PS (as well as PS2 and PS2.1) does not validate reordering of a **freeze** instruction and a release write. We note that PS does not validate reordering of non-determinism and release fences as well for the same reason. (An example obtained by replacing the release write with a release fence in the above example is a counterexample to such reorderings.)

3.8 Conclusion and Related Work

We developed a sequential model, SEQ, for reasoning about compiler optimizations in a rich weak memory model (concretely, $\text{PS}^{\textbf{na}}$, an extension of PS2.1 with non-atomic accesses), and demonstrated its applicability for compiler verification. This provides the first formal result establishing the adequacy of sequential reasoning for a full-fledged weak memory model without relying on catch-fire semantics for races, accompanied by the first non-trivial certified optimization algorithms for weak memory concurrency. While the ideas and intuitions behind the sequential reasoning are general, adequacy is specifically proved for $\text{PS}^{\textbf{na}}$. Nevertheless, we believe that SEQ can be adapted for reasoning about optimizations in other weak memory models.

Having a sequential model for compiler optimizations paves the way for future work, which has seemed to be out of reach when dealing with complicated concurrency models. This includes the extension of CompCert to weak memory concurrency (in fact, our sequential model is not far from compilers' model of C, and the simple refinement in §3.2 may well suffice), as well as of automatic tools like Alive2 [67] for SMT-based translation validation.

Our results have two main limitations. First, SEQ requires the memory layouts of the source and target to be identical, which rules out certain compiler transformations that are performed by CompCert and its extensions mentioned below (although register promotion is supported by $\mathsf{PS}^{\mathbf{na}}$). To the best of our knowledge, these are the only thread-local optimizations on non-atomics that compilers actually perform that are unsound under SEQ. Second, our refinement notion is not termination preserving (which requires fairness assumptions, possibly following [68]). Addressing these issues is left to future work.

Next, we discuss the relation to previous work.

Sequential reasoning The closest to our work is the work by Cuellar *et al.* [55] (see also [56, 69]) who develop a concurrency semantics, called "concurrent permission machine" (CPM), for CompCert that allows sequential reasoning on program optimizations. Their model has catch-fire semantics, using locks to avoid races. They also present a version of concurrent separation logic that can be used to show that a given program is race-free. While our use of permissions is inspired by these works, our results go beyond lock-based programs, and demonstrate the applicability of sequential reasoning for a significantly more involved model: (i) we handle C11-like atomic access and fences of different modes (from which locks can be implemented); (ii) the model of [55, 56] treats lock/unlock as unknown functional calls, thus forbids optimizations across locks (since they are not performed by CompCert) in contrast to our model that allows optimizations across atomics; (iii) we validate load introduction which is unsound in CPM (in fact, we found out that distinguishing read-only and write permissions, as done in [55, 56], does not suffice when write-read races are not UB, and developed the idea of written locations set (F) instead); and (iv) the target model in [55, 56] is x86-TSO, which is much simpler than the promising model studied here. All these aspects pose significant challenges in the design of the sequential model and its adequacy proof.

Certified compilation of concurrent programs Jiang *et al.* [57] presented CASCompCert, an extension of CompCert deriving certified compilation of concurrent programs from the correctness of sequential compilation, which, in particular, preserves termination. The main difference from our work is that CASCompCert targets DRF programs under sequential consistency (SC), and assumes that racy code (*e.g.*, for the implementation of locks) is confined in manually written assembly assuming x86-TSO and has race-free SC abstractions. As [55, 56], CASCompCert does not support optimizations across locked regions, reorderings of non-atomic and atomic events, and load introduction.

Another extension of CompCert, called thread-safe CompCertX, was presented by Gu *et al.* [70] in the context of the certified concurrent abstraction layers framework (CCAL). They assume SC as the underlying model, and do not support optimizations on shared non-atomics, which are ubiquitous in concurrent programming.

Earlier work extended CompCert to concurrency [71, 72, 73], for the case that both the source and the target programs have x86-TSO semantics [24] using direct TSO reasoning for the relevant optimization passes. In our terms, this assumes that all accesses are atomic with semantics stronger than release/acquire, rendering various optimizations on non-racy code unsound. Indeed, these optimizations are not performed in the optimization passes of CompCertTSO.

Verification of compiler transformations Many papers study the correctness of compiler optimizations under certain weak memory models. In particular, Burckhardt *et al.* [47] develop a denotational approach for compiler optimizations based on the rewritings performed by the target architecture; Ševčík *et al.* [48] investigates optimizations under a general catch-fire model using locks and synchronization (a.k.a. volatile) accesses; and Vafeiadis *et al.* [20] provide an extensive study (in Coq) of program transformations in the C/C++11 model [27]. The approach of [20] requires understanding of the C/C++11 model and reasoning about all possible contexts. Another important difference is that the claims in [20] are on the trace-level (represented by execution graphs) leaving implicit the connection to programs.

Based on [20], testing methods and tools for checking the correctness of compiler optimizations were developed [52, 51] and applied on randomly generated programs. Roughly speaking, these validators match (full program) source and target executions and check that the matching adheres to the set of allowed transformations.

Dodds *et al.* [63] developed a technique and a tool for verifying transformations in the fragment of C11 consisting of release/acquire atomics, non-atomics, and SC-fences. They presented a denotational framework for establishing contextual refinement and provided a push-button tool (which does not support non-atomics) using the Alloy model checker.

Program-logics-based approaches Recently, Gäher *et al.* [74] developed a separation logic (based on Iris [75]) for contextual refinement in a catch-fire model with SC atomics, allowing, in particular, optimizations involving both atomics and non-atomics. Their refinement preserves termination under fairness assumptions, and allows certain optimizations that modify the memory layout mappings. Interestingly, they considered sequential reasoning as a limitation of previous work, but, as we show, such reasoning does not have to identify atomic accesses with external function calls, and is, thus, capable of reasoning about a variety of optimizations.

Earlier work developed a rely-guarantee relational framework, which also provides means for establishing soundness of program transformations in the presence of assumptions about the environment [76, 77] . It assumes SC as the underlying model, and requires rely-guarantee reasoning for encoding, *e.g.*, data-race-freedom, versus sequential reasoning that ensures refinement under any context as we present.

Compilation scheme correctness A compilation correctness proof is not only about optimizations, and should also include the correctness of the "mapping schemes" to different architectures. In particular, the aforementioned works, including [55, 56, 57], include the correctness of mappings targeting the x86-TSO architecture. Additional proofs of the correctness of mapping schemes between more complex models appear in [38, 78]. For the promising semantics, mapping correctness was established in Coq [4] for multiple architectures (and the proof trivially generalizes to the extension with non-atomics) via IMM [31]. The latter provides an intermediate model between the programming language models and the various multicore architectures, which can be adapted to accommodate revised models on both sides.

Bibliography

[1] M. Cho, S.-H. Lee, C.-K. Hur, and O. Lahav, "Modular data-race-freedom guarantees in the promising semantics," in *PLDI*, (New York, NY, USA), pp. 867–882, ACM, 2021.

[2] M. Cho, S.-H. Lee, D. Lee, C.-K. Hur, and O. Lahav, "Sequential reasoning for optimizing compilers under weak memory concurrency," in *PLDI*, pp. 213–228, 2022.

[3] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *POPL*, (New York, NY, USA), pp. 175–189, ACM, 2017.

[4] S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis, "Promising 2.0: Global optimizations in relaxed memory concurrency," in *PLDI*, (New York, NY, USA), pp. 362–376, ACM, 2020.

[5] "Modular Data-Race-Freedom Guarantees in the Promising Semantics." `https://github.com/snu-sf/promising-ldrf-coq`, 2022.

[6] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *POPL*, (New York, NY, USA), pp. 608–621, ACM, 2016.

[7] S. V. Adve and M. D. Hill, “Weak ordering–a new definition,” in *ISCA*, (New York, NY, USA), pp. 2–14, ACM, 1990.

[8] S. Chakraborty and V. Vafeiadis, “Grounding thin-air reads with event structures,” *Proc. ACM Program. Lang.*, vol. 3, pp. 70:1–70:28, Jan. 2019.

[9] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, “Optimal stateless model checking under the release-acquire semantics,” *Proc. ACM Program. Lang.*, vol. 2, pp. 135:1–135:29, Oct. 2018.

[10] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis, “Strong logic for weak memory: Reasoning about release-acquire consistency in Iris,” in *ECOOP*, (Dagstuhl, Germany), pp. 17:1–17:29, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[11] O. Lahav and V. Vafeiadis, “Owicki-Gries reasoning for weak memory models,” in *ICALP*, (Berlin, Heidelberg), pp. 311–323, Springer, 2015.

[12] O. Lahav and R. Margalit, “Robustness against release/acquire semantics,” in *PLDI*, (New York, NY, USA), pp. 126–141, ACM, 2019.

[13] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, “Repairing sequential consistency in C/C++11,” in *PLDI*, (New York, NY, USA), pp. 618–632, ACM, 2017.

[14] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, “Effective stateless model checking for c/c++ concurrency,” *Proc. ACM Program. Lang.*, vol. 2, pp. 17:1–17:32, Dec. 2017.

[15] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, “Rustbelt meets relaxed memory,” *Proc. ACM Program. Lang.*, vol. 4, Dec. 2020.

[16] S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick, "Verifying c11 programs operationally," in *PPoPP*, (New York), pp. 355–365, ACM, 2019.

[17] A. Raad, M. Doko, L. Rožić, O. Lahav, and V. Vafeiadis, "On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.

[18] S. Dolan, K. Sivaramakrishnan, and A. Madhavapeddy, "Bounding data races in space and time," in *PLDI*, (New York, NY, USA), pp. 242–255, ACM, 2018.

[19] B. Dongol, R. Jagadeesan, and J. Riely, "Modular transactions: Bounding mixed races in space and time," in *PPoPP*, (New York, NY, USA), pp. 82–93, ACM, 2019.

[20] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, "Common compiler optimisations are invalid in the C11 memory model and what we can do about it," in *POPL*, (New York, NY, USA), pp. 209–220, ACM, 2015.

[21] P. Ou and B. Demsky, "Towards understanding the costs of avoiding out-of-thin-air results," *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018.

[22] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell, "The problem of programming language concurrency semantics," in *ESOP*, (Berlin, Heidelberg), pp. 283–307, Springer, 2015.

[23] M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty, "Modular relaxed dependencies in weak memory concurrency," in *ESOP*, (Cham), pp. 599–625, Springer, 2020.

[24] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," in *TPHOLs*, (Berlin, Heidelberg), pp. 391–407, Springer, 2009.

[25] O. Lahav, “Verification under causally consistent shared memory,” *ACM SIGLOG News*, vol. 6, pp. 43–56, Apr. 2019.

[26] O. Lahav and U. Boker, “Decidable verification under a causally consistent shared memory,” in *PLDI*, (New York, NY, USA), pp. 211–226, ACM, 2020.

[27] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, “Mathematizing C++ concurrency,” in *POPL*, (New York, NY, USA), pp. 55–66, ACM, 2011.

[28] M. Batty, A. F. Donaldson, and J. Wickerson, “Overhauling sc atomics in c11 and opencl,” in *POPL*, (New York, NY, USA), pp. 634–648, ACM, 2016.

[29] “C/C++11 mappings to processors.” `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`, 2021.

[30] Arm, “Arm a64 instruction set architecture armv8 (ddi0596 2020-12).” `https://developer.arm.com/documentation/ddi0596/2020-12`, 2020.

[31] A. Podkopaev, O. Lahav, and V. Vafeiadis, “Bridging the gap between programming languages and hardware weak memory models,” *Proc. ACM Program. Lang.*, vol. 3, pp. 69:1–69:31, Jan. 2019.

[32] M. Khiszinsky, “Cds c++ library.” `https://github.com/khizmax/libcds`, 2020.

[33] R. Jagadeesan, A. Jeffrey, and J. Riely, “Pomsets with preconditions: A simple model of relaxed memory,” *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020.

[34] J. Manson, W. Pugh, and S. V. Adve, “The java memory model,” in *POPL*, (New York, NY, USA), pp. 378–391, ACM, 2005.

[35] J. Ševčík and D. Aspinall, “On validity of program transformations in the Java memory model,” in *ECOOP*, (Berlin, Heidelberg), pp. 27–51, Springer-Verlag, 2008.

[36] J. Pichon-Pharabod and P. Sewell, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in *POPL*, (New York, NY, USA), pp. 622–633, ACM, 2016.

[37] A. Jeffrey and J. Riely, "On thin air reads: Towards an event structures model of relaxed memory," *Logical Methods in Computer Science*, vol. 15, no. 1, 2019.

[38] E. Moiseenko, A. Podkopaev, O. Lahav, O. Melkonian, and V. Vafeiadis, "Reconciling Event Structures with Modern Multiprocessors," in *ECOOP*, (Dagstuhl, Germany), pp. 5:1–5:26, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[39] J. Bender and J. Palsberg, "A formalization of java's concurrent access modes," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 142:1–142:28, 2019.

[40] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," in *POPL*, (New York, NY, USA), pp. 649–662, ACM, 2016.

[41] Y. Zhang and X. Feng, "An operational happens-before memory model," *Front. Comput. Sci.*, vol. 10, pp. 54–81, Feb. 2016.

[42] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "Drfx: An understandable, high performance, and flexible memory model for concurrent languages," *ACM Trans. Program. Lang. Syst.*, vol. 38, Sept. 2016.

[43] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems," in *ISCA*, (New York, NY, USA), pp. 161–174, ACM, 2017.

[44] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang, "Concurrent library correctness on the tso memory model," in *ESOP*, (Berlin, Heidelberg), pp. 87–107, Springer, 2012.

[45] M. Batty, M. Dodds, and A. Gotsman, "Library abstraction for c/c++ concurrency," in *POPL*, (New York, NY, USA), pp. 235–248, ACM, 2013.

[46] B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong, "On abstraction and compositionality for weak-memory linearisability," in *VMCAI*, (Cham), pp. 183–204, Springer International Publishing, 2018.

[47] S. Burckhardt, M. Musuvathi, and V. Singh, "Verifying local transformations on relaxed memory models," in *CC*, (Berlin, Heidelberg), pp. 104–123, Springer, 2010.

[48] J. Ševčík, "Safe optimisations for shared-memory concurrent programs," in *PLDI*, (New York, NY, USA), pp. 306–316, ACM, 2011.

[49] S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an llvm fragment," in *CGO*, pp. 100–110, IEEE Press, 2017.

[50] J. Pichon-Pharabod and P. Sewell, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in *POPL*, (New York, NY, USA), pp. 622–633, ACM, 2016.

[51] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the c11/c++11 memory model," in *PLDI*, (New York, NY, USA), pp. 187–196, ACM, 2013.

[52] S. Chakraborty and V. Vafeiadis, "Validating optimizations of concurrent c/c++ programs," in *CGO*, (New York, NY, USA), pp. 216–226, ACM, 2016.

[53] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[54] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, pp. 363–446, Dec. 2009.

[55] S. Cuellar, N. Giannarakis, J.-M. Madiot, W. Mansky, L. Beringer, Q. Cao, and A. W. Appel, "Compiler correctness for concurrency: from concurrent separation logic to shared-memory assembly language," Tech. Rep. TR-014-19, Department of Computer Science, Princeton University, March 2020.

[56] S. Cuellar, *Concurrent Permission Machine for Modular Proofs of Optimizing Compilers with Shared Memory Concurrency.* PhD thesis, Princeton University, 2020.

[57] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng, "Towards certified separate compilation for concurrent programs," in *PLDI*, (New York, NY, USA), pp. 111–125, ACM, 2019.

[58] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in llvm," in *PLDI*, (New York, NY, USA), pp. 633–647, ACM, 2017.

[59] K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis, "A separation logic for a promising semantics," in *ESOP*, (Cham), pp. 357–384, Springer International Publishing, 2018.

[60] "LLVM documentation: Atomic Instructions and Concurrency Guide." `https://llvm.org/docs/Atomics.html`, 2021.

[61] A. Jeffrey, J. Riely, M. Batty, S. Cooksey, I. Kaysin, and A. Podkopaev, "The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency," *Proc. ACM Program. Lang.*, vol. 6, jan 2022.

[62] "Sequential Reasoning for Optimizing Compilers Under Weak Memory Concurrency." `https://github.com/snu-sf/promising-seq-coq`, 2022.

[63] M. Dodds, M. Batty, and A. Gotsman, “Compositional verification of compiler optimisations on relaxed memory,” in *ESOP*, (Cham), pp. 1027–1055, Springer International Publishing, 2018.

[64] G. Petri, J. Vitek, and S. Jagannathan, “Cooking the books: Formalizing JMM implementation recipes,” in *ECOOP*, vol. 37, (Dagstuhl, Germany), pp. 445–469, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[65] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, “Interaction trees: Representing recursive and impure programs in coq,” *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[66] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis, “Pilsner: A compositionally verified compiler for a higher-order imperative language,” in *ICFP*, ACM, 2015.

[67] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, “Alive2: Bounded translation validation for llvm,” in *PLDI*, (New York, NY, USA), pp. 65–79, ACM, 2021.

[68] O. Lahav, E. Namakonov, J. Oberhauser, A. Podkopaev, and V. Vafeiadis, “Making weak memory models fair,” *Proc. ACM Program. Lang.*, vol. 5, oct 2021.

[69] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel, “Verified compilation for shared-memory c,” in *ESOP*, (Berlin, Heidelberg), pp. 107–127, Springer, 2014.

[70] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro, “Certified concurrent abstraction layers,” in *PLDI*, (New York, NY, USA), pp. 646–661, ACM, 2018.

[71] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, “Compcerttso: A verified compiler for relaxed-memory concurrency,” *J. ACM*, vol. 60, June 2013.

[72] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "Relaxed-memory concurrency and verified compilation," in *POPL*, (New York, NY, USA), pp. 43–54, ACM, 2011.

[73] V. Vafeiadis and F. Zappa Nardelli, "Verifying fence elimination optimisations," in *SAS*, vol. 6887 of *LNCS*, (Berlin, Heidelberg), pp. 146–162, Springer, 2011.

[74] L. Gäher, M. Sammler, S. Spies, R. Jung, H.-H. Dang, R. Krebbers, J. Kang, and D. Dreyer, "Simuliris: A separation logic framework for verifying concurrent program optimizations," *Proc. ACM Program. Lang.*, vol. 6, jan 2022.

[75] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *POPL*, (New York, NY, USA), p. 637–650, ACM, 2015.

[76] H. Liang, X. Feng, and M. Fu, "A rely-guarantee-based simulation for verifying concurrent program transformations," in *POPL*, (New York, NY, USA), pp. 455–468, ACM, 2012.

[77] H. Liang, X. Feng, and M. Fu, "Rely-guarantee-based simulation for compositional verification of concurrent program transformations," *ACM Trans. Program. Lang. Syst.*, vol. 36, Mar. 2014.

[78] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Promising Compilation to ARMv8 POP," in *ECOOP*, vol. 74, (Dagstuhl, Germany), pp. 22:1–22:28, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

초록

본 논문에서는 느슨한 메모리에서 동시성 프로그램을 구현하는 것을 쉽게 만드는 방법을 제시한다. 쓰레드들이 같은 메모리에 접근하는 동시성 프로그램에서는 하드웨어와 컴파일러의 최적화 덕분에 굉장히 비직관적인(혹은, *느슨한*) 행동이 일어난다. 이러한 느슨한 메모리 행동까지 전부 고려해서 프로그램을 올바르게 구현하는 것은 굉장히 어렵다. 우리는 느슨한 메모리 행동을 거의 고려하지 않고도 쉽고 프로그램을 구현할 수 있게 만드는 두 이론을 개발했다. 먼저, 데이터 경쟁이 없는 메모리 영역에서는 느슨한 메모리 행동이 일어나지 않는다는 부분적 무경쟁 보장을(Local Data-Race-Freedom Guarantees) 엄밀히 나타내고 증명했다. 이를 통해 데이터 경쟁을 피하는 프로그래머는 느슨한 메모리를 고려하지 않고도 동시성 프로그램을 올바르게 이해하고 작성할 수 있다. 다음으로, 동시성 프로그램을 최적화할 때 느슨한 메모리 행동과 동시성을 무시할 수 있게 해주는 이론을 개발했다. 느슨한 메모리의 복잡성을 전혀 가지지 않으며 싱글 쓰레드만 있는 것처럼 동작하는 모델 SEQ를 개발했고, 느슨한 메모리 동시성에서 올바른 최적화를 구현하기 위해서는 SEQ 위에서의 실행만 고려하면 된다는 것을 보였다. 이 결과들은 느슨한 메모리 행동을 설명하는 모델인 약속 메모리 모델을(Promising Semantics) 통해 옳다는 것이 검증되었다.

주요어: 느슨한 메모리, 실행 의미, 데이터 경쟁, 컴파일러 최적화, 컴파일러 검증
학번: 2017-29003

감사의 글

가장 먼저 제 지도교수님, 허충길 교수님께 감사드립니다. 연구를 지도해 주실 뿐만 아니라 최고의 연구 동료이셨습니다. 날카롭고 집요하게 뿌리부터 문제를 파고드시는 모습에서 많이 배웠습니다. I would like to thank Prof. Ori Lahav. Discussions with him have always led to novel and fruitful ideas. 프로그래밍 언어 분야를 제게 처음 가르쳐 주시기도 한 이광근 교수님께서는 제게 필요한 조언과 쓴소리를 아끼지 않으셨습니다.

모든 SF 연구실 동료들에게 감사합니다. 연구를 자유롭고 즐겁게 할 수 있는 환경에 있다는 것은 큰 행운이었습니다. 함께 가장 많은 연구를 한 이성환에게 특별히 감사합니다. 성환이가 없었다면 분명히 본 논문에 포함된 어떤 연구도 완성되지 못했을 것입니다. 대학원에 입학했을 때부터 쭉 저를 이끌어 준 송용주 형에게는 연구는 물론이고 다른 많은 것들을 배웠습니다.

날 응원해 준 우리 가족, 엄마와 아빠, 그리고 민호에게 감사합니다. 마지막으로 힘이 되어주는 친구들, 특히 가장 많은 시간을 함께한 최진우에게 감사합니다.