Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency

Minki Cho*
Seoul National University
Korea
minki.cho@sf.snu.ac.kr

Sung-Hwan Lee*
Seoul National University
Korea
sunghwan.lee@sf.snu.ac.kr

Dongjae Lee
Seoul National University
Korea
dongjae.lee@sf.snu.ac.kr

Chung-Kil Hur
Seoul National University
Korea
gil.hur@sf.snu.ac.kr

Ori Lahav
Tel Aviv University
Israel
orilahav@tau.ac.il

Abstract

We formally show that sequential reasoning is adequate and sufficient for establishing soundness of various compiler optimizations under weakly consistent shared-memory concurrency. Concretely, we introduce a sequential model and show that behavioral refinement in that model entails contextual refinement in the Promising Semantics model, extended with non-atomic accesses for non-racy code. This is the first work to achieve such result for a full-fledged model with a variety of C11-style concurrency features. Central to our model is the lifting of the common data-race-freedom assumption, which allows us to validate irrelevant load introduction, a transformation that is commonly performed by compilers. As a proof of concept, we develop an optimizer for a toy concurrent language, and certify it (in Coq) while relying solely on the sequential model. We believe that the proposed approach provides useful means for compiler developers and validators, as well as a solid foundation for the development of certified optimizing compilers for weakly consistent shared-memory concurrency.

CCS Concepts: • **Theory of computation** → **Concurrency**; **Operational semantics**; • **Software and its engineering** → **Semantics**; **Compilers**.

Keywords: Relaxed Memory Concurrency; Operational Semantics; Compiler Optimizations

*The first two authors contributed equally to this work.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523718

ACM Reference Format:
Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3519939.3523718

1 Introduction

Weakly consistent shared-memory semantics (a.k.a. weak, or relaxed, 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 [4, 6, 7, 14, 15, 18, 22, 32, 34, 38, 40, 41]. 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 [23, 24], whose simulation-based approach for the soundness of each optimization pass cannot accommodate complex concurrency semantics.

In this paper, 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.

[1] Informal and pen-and-paper arguments often resulted in detecting miscompilation bugs due to subtle unexpected interaction between language features; see *e.g.*, [8, Remark 7], [6, §2.2], and [5, 30].

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 1.1. 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:

$$x^{\mathsf{na}} := v\,;\ b := x^{\mathsf{na}} \ \rightsquigarrow\ x^{\mathsf{na}} := v\,;\ b := v$$

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 programs is not new, our results provide two important advantages. First, while previous work [9, 10, 16] studied a simple concurrency model based on locks or atomic blocks, the current paper 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 1.2. Continuing Example 1.1, a more interesting SLF pass eliminates reads also across other instructions:

$$x^{\mathsf{na}} := v\,;\ \alpha\ ;\ b := x^{\mathsf{na}} \ \rightsquigarrow\ x^{\mathsf{na}} := v\,;\ \alpha\ ;\ b := v$$

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 [6, §2.2]. As we show below, the model proposed in this paper allows one to analyze the soundness of this pass relying solely on a sequential model. △

Second, in contrast to prior work [9, 10, 16], 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 [2]. 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 undefined value, which is often a result of load introduction [21].)

Example 1.3. Consider an optimization pass performing loop invariant code motion (LICM) following the pattern:

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

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 §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 [8]. The latter is a recent version of the "promising semantics", a well-studied model [18, 22, 36, 37] 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 [27], to allow load introduction, which is notoriously hard to support in a relaxed memory model [**?**], racy non-atomic reads retrieve "undefined" values that can be later "frozen" into any non-deterministic value [21] (rather than invoking UB as in C/C++11).

All in all, the contributions of this paper 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 (§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 (§2) that suffices for the vast majority of optimizations (including all those involving solely

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

non-atomics), and a more refined one (§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 (§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 (§5). All results for PS2.1 in [8] are ported to PS^{na}, the extended model. We believe that PS^{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 [8]), 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 PS^{na} from (simple or refined) behavioral refinement in SEQ (§6).

Our results are fully mechanized in Coq, building on top of the existing formalization of PS2.1. Our development is available in the artifact accompanying this paper.

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 [11, 38]), 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 accesses and non-atomics, as well as roach-motel reorderings (one-sided reordering of release/acquire and non-atomics) [33], 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 (Loc^{at}) and non-atomic locations (Loc^{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 [20], except for sequentially consistent accesses which are not supported by the promising semantics.

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:

- $\sigma \to \sigma'$: 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 3 in §6 explains why we need to expose these transitions).

[3] The problem in supporting such mixing is the LOWER step of PS^{na} that modifies values of outstanding promises. We describe in [1, Appendix E], why the lower operation is needed for validating certain compiler transformations and the challenge it creates for adequacy of sequential reasoning.

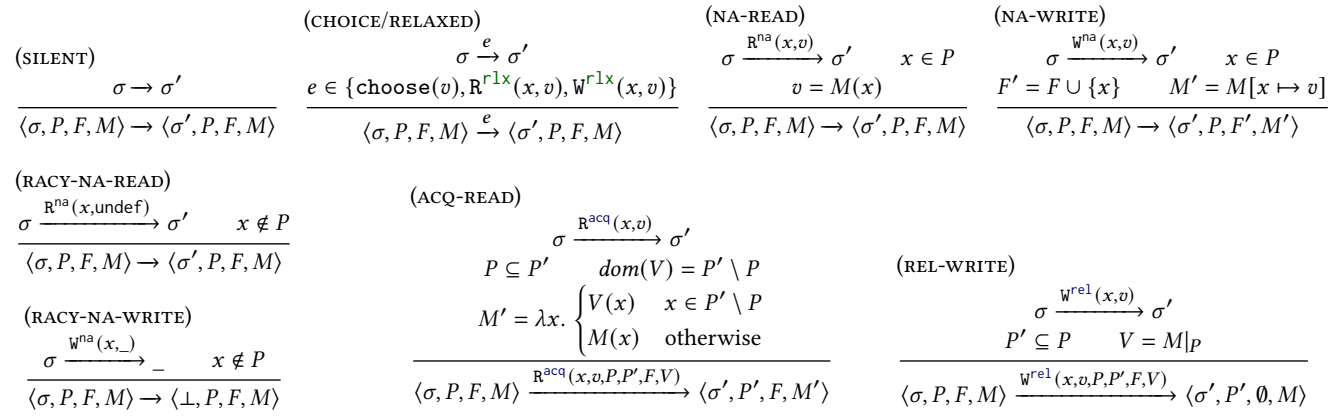


Figure 1. Transitions of SEQ

- $\sigma \xrightarrow{\texttt{R}^{o_\texttt{R}}(x,v)} \sigma'$ 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* [44], 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 1. 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 $P \subseteq \mathsf{Loc}^{\texttt{na}}$ 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 $F \subseteq \mathsf{Loc}^{\texttt{na}}$ 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 $M : \mathsf{Loc}^{\texttt{na}} \to \mathsf{Val}$ 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. 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 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 $\bot$) otherwise

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

(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 of locations (intuitively, these permissions are acquired from other threads), and (ii) gets new values (recorded in a partial function $V : \mathrm{Loc}^{\mathsf{na}} \rightharpoonup \mathrm{Val}$) 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 2.10).

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 2.1. 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:

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

$$\frac{}{\langle \sigma, P, F, M\rangle \Downarrow \langle \epsilon, r\rangle}$$

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 2.2. For a program state σ that corresponds to $x^{\mathtt{rlx}} := 1; y^{\mathsf{na}} := 2;$ **return**(3), the state $S = \langle \sigma, P, \emptyset, M\rangle$ with $y \in P$ has the following behaviors in SEQ: $\langle \epsilon, \mathtt{prt}(\emptyset)\rangle$, $\langle \mathtt{W}^{\mathtt{rlx}}(x, 1), \mathtt{prt}(\emptyset)\rangle$, $\langle \mathtt{W}^{\mathtt{rlx}}(x, 1), \mathtt{prt}(\{y\})\rangle$, and the terminating behavior $\langle \mathtt{W}^{\mathtt{rlx}}(x, 1), \mathtt{trm}(3, \{y\}, M[y \mapsto 2])\rangle$. If $y \notin P$, then $\langle \mathtt{W}^{\mathtt{rlx}}(x, 1), \bot\rangle$ is the only terminating behavior. △

Next, we present the (first) notion of behavioral refinement between programs in SEQ. As standard, one may start by requiring that every behavior $\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle$ 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 $\langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ that is "less committed" than $\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle$ (denoted $\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$): the source may return `undef` when the target returns a normal value ($v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}}$), end with "less defined" memory ($M_{\mathrm{tgt}} \sqsubseteq M_{\mathrm{src}}$), and write to more non-atomic locations ($F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}}$). 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 2.3. The relation $\sqsubseteq$ on transition labels, traces, and behaviors is given by:

1. Transition labels:

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

$$\frac{v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}} \qquad F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}} \qquad V_{\mathrm{tgt}} \sqsubseteq V_{\mathrm{src}}}{\mathtt{W}^{\mathtt{rel}}(x, v_{\mathrm{tgt}}, P, P', F_{\mathrm{tgt}}, V_{\mathrm{tgt}}) \sqsubseteq \mathtt{W}^{\mathtt{rel}}(x, v_{\mathrm{src}}, P, P', F_{\mathrm{src}}, V_{\mathrm{src}})}$$

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

$$\frac{tr_{\mathrm{tgt}} \sqsubseteq tr_{\mathrm{src}} \quad v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}} \quad F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}} \quad M_{\mathrm{tgt}} \sqsubseteq M_{\mathrm{src}}}{\langle tr_{\mathrm{tgt}}, \mathtt{trm}(v_{\mathrm{tgt}}, F_{\mathrm{tgt}}, M_{\mathrm{tgt}})\rangle \sqsubseteq \langle tr_{\mathrm{src}}, \mathtt{trm}(v_{\mathrm{src}}, F_{\mathrm{src}}, M_{\mathrm{src}})\rangle}$$

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

Definition 2.4. We write $S_{\mathrm{tgt}} \sqsubseteq S_{\mathrm{src}}$ if $S_{\mathrm{tgt}} \Downarrow \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle$ implies that $S_{\mathrm{src}} \Downarrow \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ for some behavior $\langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ such that $\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$. A program state σ_{tgt} *behaviorally refines* a program state σ_{src}, denoted by $\sigma_{\mathrm{tgt}} \sqsubseteq \sigma_{\mathrm{src}}$, if $\langle \sigma_{\mathrm{tgt}}, P, F, M\rangle \sqsubseteq \langle \sigma_{\mathrm{src}}, P, F, M\rangle$ for every P, F, M.

Next, we present a sequence of examples demonstrating several subtleties in the above definitions. In these examples, when writing $prog_{\mathrm{src}} \rightsquigarrow prog_{\mathrm{tgt}}$ for two code snippets $prog_{\mathrm{src}}$ and $prog_{\mathrm{tgt}}$, we mean that for any (sequential) context C, the state σ_{tgt} that corresponds to $C[prog_{\mathrm{tgt}}]$ (with some

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

initial register file) behaviorally refines the state σ_{src} that runs $C[prog_{\text{src}}]$ (with the same initial register file). We write $prog_{\text{src}} \not\rightsquigarrow prog_{\text{tgt}}$ for the negation of $prog_{\text{src}} \rightsquigarrow prog_{\text{tgt}}$.

Remark 2. 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 $prog_{\text{src}} \rightsquigarrow prog_{\text{tgt}}$ 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 2.5 (Reordering of non-atomics)**.** Non-atomic accesses to different locations can be freely reordered in SEQ, *e.g.*, $a := x^{\text{na}}; y^{\text{na}} := v \rightsquigarrow y^{\text{na}} := v; a := x^{\text{na}}$ where $x \neq y$. Consider a general context C, and let σ_{src} and σ_{tgt} be the program states that correspond to $C[a := x^{\text{na}}; y^{\text{na}} := v]$ and $C[y^{\text{na}} := v; a := x^{\text{na}}]$, 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 $a := x^{\text{na}}; x^{\text{na}} := 1 \not\rightsquigarrow x^{\text{na}} := 1; a := x^{\text{na}}$. Indeed, for the context $C = \cdot; \mathbf{return}(a)$, we have $\langle \sigma_{\text{tgt}}, P, F, M\rangle \Downarrow \langle \epsilon, \mathtt{trm}(1, F \cup \{x\}, M[x \mapsto 1])\rangle$ 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 $\langle \epsilon, \mathtt{trm}(2, F \cup \{x\}, M[x \mapsto 1])\rangle$. △

Example 2.6 (Eliminations of non-atomics)**.** Behavioral refinement holds for the following pairs, in which a non-atomic access is eliminated:

$$(i) \qquad x^{\text{na}} := v; x^{\text{na}} := v' \rightsquigarrow x^{\text{na}} := v'$$

$$(ii) \qquad x^{\text{na}} := v; a := x^{\text{na}} \rightsquigarrow x^{\text{na}} := v; a := v$$

$$(iii) \qquad a := x^{\text{na}}; b := x^{\text{na}} \rightsquigarrow a := x^{\text{na}}; b := a$$

$$(iv) \qquad a := x^{\text{na}}; x^{\text{na}} := a \rightsquigarrow a := x^{\text{na}}$$

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 ($F_{\text{tgt}} \subset F_{\text{src}}$), 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:

$$a := x^{\text{na}}; \textbf{if } a \neq v \textbf{ then } x^{\text{na}} := v \not\rightsquigarrow a := x^{\text{na}}; x^{\text{na}} := v$$

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

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 2.7 (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:

$$\textbf{while } (\ldots) \textbf{ do } \{\ldots\}; x^{\text{na}} := v \not\rightsquigarrow x^{\text{na}} := v; \textbf{while } (\ldots) \textbf{ do } \{\ldots\}$$

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 $F_{\text{tgt}} \subseteq F_{\text{src}}$:

$$\begin{array}{l} a := x^{\text{na}}; \\ \textbf{if } a \neq 1 \textbf{ then } x^{\text{na}} := 1; \\ \textbf{while } (\ldots) \textbf{ do } \{\ldots\}; x^{\text{na}} := 2 \end{array} \not\rightsquigarrow \begin{array}{l} a := x^{\text{na}}; \\ \textbf{if } a \neq 1 \textbf{ then } x^{\text{na}} := 1; \\ x^{\text{na}} := 2; \textbf{while } (\ldots) \textbf{ do } \{\ldots\} \end{array}$$

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 $\langle \_, \mathtt{prt}(F)\rangle$. Indeed, when starting with permission on x and $M(x) = 1$, the target generates $\langle \epsilon, \mathtt{prt}(\{x\})\rangle$, 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:

$$\textbf{while } (\ldots) \textbf{ do } \{\ldots\}; a := x^{\text{na}} \rightsquigarrow a := x^{\text{na}}; \textbf{while } (\ldots) \textbf{ do } \{\ldots\}$$

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 2.8 (Unused load elimination and introduction)**.** The transformations that eliminate/introduce an unused load $a := x^{\text{na}} \rightsquigarrow \textbf{skip}$ and $\textbf{skip} \rightsquigarrow a := x^{\text{na}}$ 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 2.9 (Reordering of atomics and non-atomics)**.** Reordering of atomic and non-atomic accesses follows the "roach-motel" principle. The following are forbidden:

$$(i) \quad a := x^{\mathsf{acq}}; y^{\mathsf{na}} := v \not\rightsquigarrow y^{\mathsf{na}} := v; a := x^{\mathsf{acq}}$$

$$(ii) \quad y^{\mathsf{na}} := v'; x^{\mathsf{rel}} := v \not\rightsquigarrow x^{\mathsf{rel}} := v; y^{\mathsf{na}} := v'$$

$$(iii) \quad a := x^{\mathsf{acq}}; b := y^{\mathsf{na}} \not\rightsquigarrow b := y^{\mathsf{na}}; a := x^{\mathsf{acq}}$$

$$(iv) \quad a := y^{\mathsf{na}}; x^{\mathsf{rel}} := v \not\rightsquigarrow x^{\mathsf{rel}} := v; a := y^{\mathsf{na}}$$

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 $\langle \mathtt{R}^{\mathsf{acq}}(\_), \bot \rangle$ 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 $C = \cdot; \mathbf{return}(b)$, if a permission on y is gained by the acquire read from x, and we start with $y \notin P$ (and $M(y) \neq \mathsf{undef}$), the target generates a behavior of the form $\langle \mathtt{R}^{\mathsf{acq}}(x, \_, P, P \cup \{y\}, \_, \_), \mathtt{trm}(\mathsf{undef}, \_, \_) \rangle$, whereas the source cannot perform racy read on y.

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

Next, the following converses of the above are validated:

$$(i') \quad y^{\mathsf{na}} := v; a := x^{\mathsf{acq}} \rightsquigarrow a := x^{\mathsf{acq}}; y^{\mathsf{na}} := v$$

$$(iii') \quad b := y^{\mathsf{na}}; a := x^{\mathsf{acq}} \rightsquigarrow a := x^{\mathsf{acq}}; b := y^{\mathsf{na}}$$

$$(iv') \quad x^{\mathsf{rel}} := v; a := y^{\mathsf{na}} \rightsquigarrow a := y^{\mathsf{na}}; x^{\mathsf{rel}} := v$$

For (i') we use the fact that acquire transitions of the target can be matched by acquire transitions of the source annotated with $F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}}$ (since we may have $y \in F_{\mathrm{src}}$ but not $y \in F_{\mathrm{tgt}}$ when performing the acquire), as well as the fact that the source may invoke UB earlier than the target (in particular, $\langle \mathtt{R}^{\mathsf{acq}}(\_), \bot \rangle \sqsubseteq \langle \epsilon, \bot \rangle$). 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. △

Example 2.10. Stores cannot be introduced even if they already occur before a release write:

$$x^{\mathsf{na}} := v; y^{\mathsf{rel}} := 1 \not\rightsquigarrow x^{\mathsf{na}} := v; y^{\mathsf{rel}} := 1; x^{\mathsf{na}} := v$$

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

$$x^{\mathsf{na}} := v; \ \alpha\ ; b := x^{\mathsf{na}} \rightsquigarrow x^{\mathsf{na}} := v; \ \alpha\ ; b := v$$

where $\alpha \in \{a := y^{\mathsf{rlx}}, y^{\mathsf{rlx}} := v', a := y^{\mathsf{acq}}, y^{\mathsf{rel}} := v'\}$. 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 $b := x^{\mathsf{na}}$, or, if α corresponds to a release write that transferred the permission on x, the source will read undef, and we have $v \sqsubseteq \mathsf{undef}$. In any case, the source matches every behavior of the target. △

Example 2.12. Reads *cannot* be eliminated after writes across *release-acquire pairs*:

$$x^{\mathsf{na}} := v; y^{\mathsf{rel}} := v'; a := z^{\mathsf{acq}}; b := x^{\mathsf{na}} \not\rightsquigarrow x^{\mathsf{na}} := v; y^{\mathsf{rel}} := v'; a := z^{\mathsf{acq}}; b := v$$

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. △

3 Advanced Behavior Refinement

As we show in §6, 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 §6 provides contextual refinement, one may mix and match—prove most optimization passes using the simple notion in §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:

$$a := x^{\mathsf{rlx}}; y^{\mathsf{na}} := v \rightsquigarrow y^{\mathsf{na}} := v; a := x^{\mathsf{rlx}}$$

Indeed, the reasoning in Example 2.9 (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.*, $\alpha\,;\ a := {}^1\!/_0 \rightsquigarrow a := {}^1\!/_0\,;\ \alpha$. As the next example shows, in concurrent settings this reordering must be invalidated if α contains an acquire read.

Example 3.1. Consider the following optimizations:

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

First, we perform the (unsound) reordering of an acquire read with UB-invoking operation. Then, a sequence of standard optimizations (start with $b := {}^1\!/_0 \rightsquigarrow y^{\mathtt{rlx}} := 1\,;\ b := {}^1\!/_0$, 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 $a := x^{\mathtt{acq}}\,;\ b := {}^1\!/_0 \rightsquigarrow b := {}^1\!/_0\,;\ a := x^{\mathtt{acq}}$ and $a := x^{\mathtt{acq}}\,;\ y^{\mathtt{na}} := v \rightsquigarrow y^{\mathtt{na}} := v\,;\ a := x^{\mathtt{acq}}$: 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:

$$\begin{array}{l} a := x^{\mathtt{rlx}}\,; \textbf{ if } a = 1 \textbf{ then } b := {}^1\!/_0\,; \\ \textbf{while } (\ldots) \textbf{ do } \{\ldots\} \end{array} \not\rightsquigarrow \begin{array}{l} b := {}^1\!/_0\,; a := x^{\mathtt{rlx}}\,; \\ \textbf{while } (\ldots) \textbf{ do } \{\ldots\} \end{array}$$

Without additional restrictions, since we allow any behavior of the target to be matched with $\langle \mathtt{R}^{\mathtt{rlx}}(x, 1), \bot \rangle$ 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 ($a := x^{\mathtt{rlx}}\,;\ y^{\mathtt{na}} := v \rightsquigarrow y^{\mathtt{na}} := v\,;\ a := x^{\mathtt{rlx}}$), 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:

$$x^{\mathtt{rel}} := v\,;\ y^{\mathtt{na}} := v' \rightsquigarrow y^{\mathtt{na}} := v'\,;\ x^{\mathtt{rel}} := v$$

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 target's label $\mathtt{W}^{\mathtt{rel}}(x, v, P, P', F_{\text{tgt}}, V_{\text{tgt}})$ must have $y \in F_{\text{tgt}}$ and $V_{\text{tgt}}(y) = v'$, whereas the source is confined by the initial state (which may have $y \notin F_{\text{src}}$ or $V_{\text{src}}(y) \neq v'$). 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

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

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

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

$$\frac{e_{\mathrm{tgt}} \sqsubseteq e_{\mathrm{src}} \qquad \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \qquad e_{\mathrm{tgt}} = \mathtt{R}^{\mathtt{rlx}}(\_,\_) \vee e_{\mathrm{tgt}} = \mathtt{W}^{\mathtt{rlx}}(\_,\_)}{\langle e_{\mathrm{tgt}} \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle e_{\mathrm{src}} \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle}\ (\textsc{beh-rlx})$$

$$\frac{\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_\emptyset \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \qquad F_{\mathrm{tgt}} \cup R \subseteq F_{\mathrm{src}}}{\langle \mathtt{R}^{\mathtt{acq}}(x,v,P,P',F_{\mathrm{tgt}},V) \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle \mathtt{R}^{\mathtt{acq}}(x,v,P,P',F_{\mathrm{src}},V) \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle}\ (\textsc{beh-acq-read})$$

$$\frac{v_{\mathrm{tgt}} \sqsubseteq v_{\mathrm{src}} \qquad \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_{R'} \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle \qquad R' = (R \setminus F_{\mathrm{src}}) \cup (F_{\mathrm{tgt}} \setminus F_{\mathrm{src}}) \cup \{y \in \mathsf{Loc}^{\mathsf{na}} \mid V_{\mathrm{tgt}}(y) \not\sqsubseteq V_{\mathrm{src}}(y)\}}{\langle \mathtt{W}^{\mathtt{rel}}(x,v_{\mathrm{tgt}},P,P',F_{\mathrm{tgt}},V_{\mathrm{tgt}}) \cdot tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_R \langle \mathtt{W}^{\mathtt{rel}}(x,v_{\mathrm{src}},P,P',F_{\mathrm{src}},V_{\mathrm{src}}) \cdot tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle}\ (\textsc{beh-rel-write})$$

Figure 2. Behavioral refinement up to a commitment set $R \subseteq \mathsf{Loc}^{\mathsf{na}}$

(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 3.2. 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 $x \in \mathsf{Loc}^{\mathsf{at}}$, $v \in \mathsf{Val}$, and $P \subseteq \mathsf{Loc}^{\mathsf{na}}$, 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. 2. It modifies the one in Def. 2.3 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 $\mathtt{W}^{\mathtt{rel}}(x,v,P,P',F_{\mathrm{tgt}},V_{\mathrm{tgt}})$ is added to the target's trace, we compare it to the matching one by the source $\mathtt{W}^{\mathtt{rel}}(x,v,P,P',F_{\mathrm{src}},V_{\mathrm{src}})$, 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 ($y \in F_{\mathrm{tgt}} \setminus F_{\mathrm{src}}$); (ii) have a value in the target memory that does not refine the value of the source ($V_{\mathrm{tgt}}(y) \not\sqsubseteq V_{\mathrm{src}}(y)$); or (iii) were included in the previous commitment set and not written yet by the source ($y \in R \setminus F_{\mathrm{src}}$). Upon termination or acquire read, in addition to $F_{\mathrm{tgt}} \subseteq F_{\mathrm{src}}$, we require that all outstanding commitments were fulfilled by the source ($R \subseteq F_{\mathrm{src}}$). 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 3.3. A program state σ_{tgt} *weakly behaviorally refines* a program state σ_{src}, denoted by $\sigma_{\mathrm{tgt}} \sqsubseteq_{\mathrm{w}} \sigma_{\mathrm{src}}$, if for every oracle Ω, if $\langle \sigma_{\mathrm{tgt}}, P, F, M\rangle \Downarrow \langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle$ and $tr_{\mathrm{tgt}} \in \mathsf{Tr}(\Omega)$, then $\langle \sigma_{\mathrm{src}}, P, F, M\rangle \Downarrow \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ for some $\langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ such that $\langle tr_{\mathrm{tgt}}, r_{\mathrm{tgt}}\rangle \sqsubseteq_\emptyset \langle tr_{\mathrm{src}}, r_{\mathrm{src}}\rangle$ and $tr_{\mathrm{src}} \in \mathsf{Tr}(\Omega)$.

Proposition 3.4. $\sigma_{\mathrm{tgt}} \sqsubseteq \sigma_{\mathrm{src}} \Rightarrow \sigma_{\mathrm{tgt}} \sqsubseteq_{\mathrm{w}} \sigma_{\mathrm{src}}$.

Example 3.5 (Overwritten store elimination across atomics)**.** Consider the elimination of a write after another write to the same location across an atomic access:

$$x^{\mathsf{na}} := v\,;\ \alpha\ ;\ x^{\mathsf{na}} := v' \ \rightsquigarrow\ \alpha\ ;\ x^{\mathsf{na}} := v'$$

where $\alpha \in \{b := y^{\mathtt{rlx}}, y^{\mathtt{rlx}} := v_y, b := y^{\mathtt{acq}}, y^{\mathtt{rel}} := v_y\}$. The three cases except for $\alpha = y^{\mathtt{rel}} := v_y$ 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 α is a release write should be also considered sound,[6] since, roughly speaking, other threads that can observe $x^{\mathsf{na}} := v$ can always also observe $x^{\mathsf{na}} := v'$ instead. (In particular, this optimization is sound in the promising semantics.) Nevertheless, the simple refinement notion in §2 invalidates this optimization ($\sigma_{\mathrm{tgt}} \not\sqsubseteq \sigma_{\mathrm{src}}$): 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 $\sigma_{\mathrm{tgt}} \sqsubseteq_{\mathrm{w}} \sigma_{\mathrm{src}}$. In particular, consider the empty context, and let $\mathtt{rel}(P,P',F,u) \triangleq \mathtt{W}^{\mathtt{rel}}(y,v_y,P,P',F,M[x \mapsto u])$. If we start with permission on x and do not release it, then for $r = \mathtt{trm}(\mathtt{unit}, \{x\}, M[x \mapsto v'])$,

$$\langle \mathtt{rel}(\{x\},\{x\},\{x\},v), r\rangle \sqsubseteq_\emptyset \langle \mathtt{rel}(\{x\},\{x\},\emptyset,M(x)), r\rangle$$

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

$$\langle \mathtt{rel}(\{x\},\emptyset,\{x\},v), \bot\rangle \sqsubseteq_\emptyset \langle \mathtt{rel}(\{x\},\emptyset,\emptyset,M(x)), \bot\rangle$$

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:

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

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

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

Figure 3. Store-to-load forwarding analysis

4 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 §6 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 $\quad x^{\mathsf{na}} := v\,;\ \alpha\ ;\ b := x^{\mathsf{na}} \rightsquigarrow x^{\mathsf{na}} := v\,;\ \alpha\ ;\ b := v$

LLF $\quad a := x^{\mathsf{na}}\,;\ \beta\ ;\ b := x^{\mathsf{na}} \rightsquigarrow a := x^{\mathsf{na}}\,;\ \beta\ ;\ b := a$

DSE $\quad x^{\mathsf{na}} := a\,;\ \gamma\ ;\ x^{\mathsf{na}} := b \rightsquigarrow \mathbf{skip}\,;\ \gamma\ ;\ x^{\mathsf{na}} := b$

LICM $\quad \mathbf{while}\ (\ldots)\ \mathbf{do}\ \{\beta_1\ ;\ a := x^{\mathsf{na}}\,;\ \beta_2\} \rightsquigarrow c := x^{\mathsf{na}}\,;\ \mathbf{while}\ (\ldots)\ \mathbf{do}\ \{\beta_1\ ;\ a := c\,;\ \beta_2\}$

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.

Next, we focus on the SLF pass and describe the analysis and optimization in detail. The other passes are described in [1, Appendix D]. Figure 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, the abstract state of x transitions to $\circ(v)$ for a non-atomic write $x^{\mathsf{na}} := v$; $\circ(v)$ 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 $a := x^{\mathsf{na}}$ 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 4 describes how the analysis and optimization work for a concrete program example.

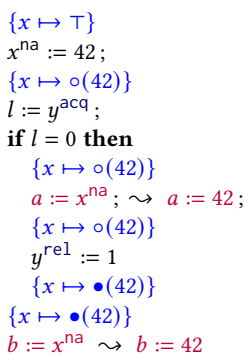
The verification of the passes is executed by (i) establish the soundness of each analysis; and (ii) from the soundness, derive a simulation in SEQ between the source and target codes. The simulation relation in SEQ (given in [1, Appendix A]) ensures advanced behavioral refinement as defined in §3.[8] This verification strategy follows the standard approach of CompCert, and, importantly, the optimizer is fully verified in Coq relying solely on sequential reasoning.

5 Non-atomics in the Promising Semantics

We present the extension of PS2.1 with non-atomic accesses, which we denote by $\mathrm{PS}^{\mathsf{na}}$. 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 (§2), we allow the mixing of atomic and non-atomic accesses to the same location (so we assume one set Loc of locations), which means that a race may involve only one non-atomic

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

[8] In fact, this is the same simulation used in the adequacy proof (see §6), so the optimizer correctness argument is independent of behavioral refinement in SEQ—it goes directly from simulation in SEQ to simulation in $\mathrm{PS}^{\mathsf{na}}$.



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 $x^{\mathsf{na}} := 42$ 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 $l := y^{\mathsf{acq}}$, 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, $a := x^{\mathsf{na}}$ will load 42 from the memory as the abstract state indicates. For the next instruction, $y^{\mathtt{rel}} := 1$, 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, $b := x^{\mathsf{na}}$, 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 4. An example optimization by SLF including the underlying analysis in SEQ

access. As for SEQ, we only present a simplified fragment of the full model omitting fences, RMWs, release sequences, and system calls, which are all covered by our Coq formalization.

Roughly, in the promising model the memory is a set of *timestamped messages* capturing all previous writes. Each thread maintains a *view*, pointing to the latest message the thread has observed for each location, used to restrict future reads/writes of the thread. *Promises* are used to allow read-write reordering—a thread may promise to perform a write in the future and add a message to memory before the write is being executed. To avoid "causality cycles" (a.k.a. thin-air behaviors) every step has to be accompanied by *certification*: by running *alone* the thread has to be able to fulfill its outstanding promises.

Figure 5 presents the thread configuration steps and the machine steps. Next, we discuss the new parts, highlighted in the figure. We refer the reader to [8, 18, 22] for explanations of the atomics fragment of PS$^{\mathsf{na}}$, which is identical to PS2.1.

Thread configuration steps. 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 [1, Appendix B]).

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

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$^{\mathsf{na}}$ (LOWER), which allows threads to modify their own promises, also allows to change a non-`undef` value of a promise to `undef` (see [1, Appendix E]).

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

$$\begin{array}{l} a := x^{\mathsf{na}}; \quad /\!/\mathtt{undef} \\ y^{\mathtt{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := y^{\mathtt{rlx}}; \\ \textbf{if } b = 1 \textbf{ then} \\ \quad x^{\mathsf{na}} := 1 \end{array}$$

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$^{\mathsf{na}}$ 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

$v \in \mathsf{Val}$	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
$o_{\mathtt{R}} \in \{\mathsf{na}, \mathtt{rlx}, \mathtt{acq}\}$	read access mode	$m = \langle x @ t, v, V \rangle \in \mathsf{Msg}$	message	$\langle T, M \rangle$	thread configuration
$o_{\mathtt{W}} \in \{\mathsf{na}, \mathtt{rlx}, \mathtt{rel}\}$	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)
$$\frac{\begin{array}{c} \langle P, M\rangle \xrightarrow{m_1} \ldots \xrightarrow{m_n} \langle P_n, M_n\rangle \xrightarrow{m} \langle P', M'\rangle \qquad n \geq 0 \\ m_1, \ldots, m_n \in \mathsf{Msg} \cup \mathsf{NAMsg} \\ m_1.\mathtt{loc} = \ldots = m_n.\mathtt{loc} = m.\mathtt{loc} \qquad t < m_1.\mathtt{t}, \ldots, m_n.\mathtt{t} < m.\mathtt{t} \\ m_1.\mathtt{view} = \ldots = m_n.\mathtt{view} = m.\mathtt{view} = \bot \end{array}}{\langle P, M\rangle \xrightarrow{t,m}_{\mathsf{na}} \langle P', M'\rangle}$$

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

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

(RACY-READ)
$$\frac{\sigma \xrightarrow{\mathtt{R}^{o_{\mathtt{R}}}(x,\mathsf{undef})} \sigma' \qquad \langle V, P, M\rangle \text{ is racy on } x \text{ with } o_{\mathtt{R}}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V, P\rangle, M\rangle}$$

(RACY-WRITE)
$$\frac{\sigma \xrightarrow{\mathtt{W}^{o_{\mathtt{W}}}(x,\_)} \sigma' \qquad \langle V, P, M\rangle \text{ is racy on } x \text{ with } o_{\mathtt{W}} \qquad \forall m \in P.\ V(m.\mathtt{loc}) < m.\mathtt{t}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \bot, V, \emptyset\rangle, M\rangle}$$

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

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

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

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

(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 5. Transitions of $\mathrm{PS}^{\mathsf{na}}$ (differences w.r.t. the corresponding fragment of PS2.1 are highlighted)

hardware follows from the soundness of this strengthening and of the mapping PS2.1 to hardware as shown in [8, 22].

Behavioral refinement. A behavior and behavioral refinement in $\mathrm{PS}^{\mathsf{na}}$ are defined as follows.[10]

Definition 5.2. A *behavior* (in $\mathrm{PS}^{\mathsf{na}}$) 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 machine state $\langle \mathcal{T}, M\rangle$ *generates* a behavior r, denoted by $\langle \mathcal{T}, M\rangle \Downarrow r$:

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

We write $r_{\mathsf{tgt}} \sqsubseteq r_{\mathsf{src}}$ if either $r_{\mathsf{src}} = \bot$ or $\forall \pi.\ r_{\mathsf{tgt}}(\pi) \sqsubseteq r_{\mathsf{src}}(\pi)$.

Definition 5.3. A concurrent program state $\sigma^1_{\mathsf{tgt}} \| \ldots \| \sigma^n_{\mathsf{tgt}}$ *behaviorally refines* a concurrent program state $\sigma^1_{\mathsf{src}} \| \ldots \| \sigma^n_{\mathsf{src}}$, denoted by $\sigma^1_{\mathsf{tgt}} \| \ldots \| \sigma^n_{\mathsf{tgt}} \sqsubseteq_{\mathrm{PS}^{\mathsf{na}}} \sigma^1_{\mathsf{src}} \| \ldots \| \sigma^n_{\mathsf{src}}$, if whenever we have $\langle \lambda \pi. \langle \sigma^\pi_{\mathsf{tgt}}, V_{\mathsf{init}}, \emptyset\rangle, M_{\mathsf{init}}\rangle \Downarrow r_{\mathsf{tgt}}$, there exists r_{src} such that $r_{\mathsf{tgt}} \sqsubseteq r_{\mathsf{src}}$ and $\langle \lambda \pi. \langle \sigma^\pi_{\mathsf{src}}, V_{\mathsf{init}}, \emptyset\rangle, M_{\mathsf{init}}\rangle \Downarrow r_{\mathsf{src}}$. (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 $x \in \mathsf{Loc}$.)

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

6 Adequacy of Sequential Reasoning

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

Definition 6.1. A program state σ is *deterministic* if for every σ_0 reachable from σ (*i.e.*, $\sigma \to^* \sigma_0$), 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 6.2 (Adequacy). *If $\sigma_{\mathsf{tgt}} \sqsubseteq_{\mathrm{W}} \sigma_{\mathsf{src}}$ (Def. 3.3) and σ_{src} is deterministic, then $\sigma_{\mathsf{tgt}} \| \sigma_1 \| \ldots \| \sigma_n \sqsubseteq_{\mathrm{PS}^{\mathsf{na}}} \sigma_{\mathsf{src}} \| \sigma_1 \| \ldots \| \sigma_n$ (Def. 5.3) for any programs $\sigma_1, \ldots, \sigma_n$.*

To prove this theorem, we first show that $\sigma_{\mathsf{tgt}} \sqsubseteq_{\mathrm{W}} \sigma_{\mathsf{src}}$ implies the existence of a simulation relation between the source and target in SEQ (detailed in [1, Appendix A]). Then, we show that a simulation in SEQ implies the existence of a simulation in $\mathrm{PS}^{\mathsf{na}}$. 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.

Remark 3. Theorem 6.2 does not hold without the determinism premise (see [1, Appendix C] for an example). This stems from a drawback of the promising semantics (rather than due to SEQ) that we encountered while developing SEQ. Concretely, the promising semantics disallows the reordering of an internal non-deterministic choice 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 leave to future work to improve the promising semantics to allow this reordering, which will allow one to remove the `choose(_)` labels from SEQ.

7 Conclusion and Related Work

We developed a sequential model, SEQ, for reasoning about compiler optimizations in a rich weak memory model (concretely, PS^{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 PS^{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 §2 may well suffice), as well as of automatic tools like Alive2 [28] 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 PS^{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 [19]). 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. [10] (see also [3, 9]) 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 [9, 10] 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

[9, 10], 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 [9, 10] 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. [16] 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 [9, 10], 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. [13] 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 [39, 42, 43], for the case that both the source and the target programs have x86-TSO semantics [31] 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. [4] develop a denotational approach for compiler optimizations based on the rewritings performed by the target architecture; Ševčík [40] investigates optimizations under a general catch-fire model using locks and synchronization (a.k.a. volatile) accesses; and Vafeiadis et al. [38] provide an extensive study (in Coq) of program transformations in the C/C++11 model [2]. The approach of [38] requires understanding of the C/C++11 model and reasoning about all possible contexts. Another important difference is that the claims in [38] are on the trace-level (represented by execution graphs) leaving implicit the connection to programs.

Based on [38], testing methods and tools for checking the correctness of compiler optimizations were developed [5, 30] 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. [11] 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. [12] developed a separation logic (based on Iris [17]) 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 [25, 26] . 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 [9, 10, 16], include the correctness of mappings targeting the x86-TSO architecture. Additional proofs of the correctness of mapping schemes between more complex models appear in [29, 35]. For the promising semantics, mapping correctness was established in Coq [22] for multiple architectures (and the proof trivially generalizes to the extension with non-atomics) via IMM [36]. 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.

Acknowledgments

We thank the anonymous PLDI reviewers for their helpful feedback. Chung-Kil Hur is the corresponding author. Minki Cho, Sung-Hwan Lee, Dongjae Lee, and Chung-Kil Hur were supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03. Ori Lahav was supported by the Israel Science Foundation (grant number 1566/18) and by the Alon Young Faculty Fellowship. This research was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811).

References

[1] 2022. Coq development and supplementary material for this paper. https://sf.snu.ac.kr/promising-seq

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

[3] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP*. Springer, Berlin, Heidelberg, 107–127. https://doi.org/10.1007/978-3-642-54833-8_7

[4] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying Local Transformations on Relaxed Memory Models. In *CC*. Springer, Berlin, Heidelberg, 104–123. https://doi.org/10.1007/978-3-642-11970-5_7

[5] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *CGO*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2854038.2854051

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

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

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

[9] Santiago Cuellar. 2020. *Concurrent Permission Machine for Modular Proofs of Optimizing Compilers with Shared Memory Concurrency*. Ph. D. Dissertation. Princeton University.

[10] Santiago Cuellar, Nick Giannarakis, Jean-Marie Madiot, William Mansky, Lennart Beringer, Qinxiang Cao, and Andrew W Appel. 2020. *Compiler Correctness for Concurrency: from concurrent separation logic to shared-memory assembly language*. Technical Report TR-014-19. Department of Computer Science, Princeton University.

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

[12] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (jan 2022), 31 pages. https://doi.org/10.1145/3498689

[13] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *PLDI*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

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

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

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

[17] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

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

[19] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making Weak Memory Models Fair. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 98 (oct 2021), 27 pages. https://doi.org/10.1145/3485475

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

[21] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. ACM, New York, NY, USA, 633–647. https://doi.org/10.1145/3062341.3062343

[22] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI*. ACM, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

[23] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[24] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[25] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In *POPL*. ACM, New York, NY, USA, 455–468. https://doi.org/10.1145/2103656.2103711

[26] Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 3 (Mar. 2014), 55 pages. https://doi.org/10.1145/2576235

[27] LLVM documentation: Atomic Instructions and Concurrency Guide 2021. Retrieved Novermber, 2021 from https://llvm.org/docs/Atomics.html

[28] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *PLDI*. ACM, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[29] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2020. Reconciling Event Structures with Modern Multiprocessors. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.5

[30] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *PLDI*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2491956.2491967

[31] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. Springer, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[32] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP*. Springer, Cham, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

[33] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. In *ECOOP*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 445–469. https://doi.org/10.4230/LIPIcs.ECOOP.2015.445

[34] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids

Thin-Air Executions. In *POPL*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

[35] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *ECOOP*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.22

[36] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

[37] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP*. Springer International Publishing, Cham, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[38] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *POPL*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

[39] Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *SAS (LNCS, Vol. 6887)*. Springer, Berlin, Heidelberg, 146–162. https://doi.org/10.1007/978-3-642-23702-7_14

[40] Jaroslav Ševčík. 2011. Safe Optimisations for Shared-memory Concurrent Programs. In *PLDI*. ACM, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

[41] Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. Springer-Verlag, Berlin, Heidelberg, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

[42] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *POPL*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/1926385.1926393

[43] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

[44] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

$$
\begin{array}{l}
\langle \sigma_{\text{src}}, F_{\text{src}}, M_{\text{src}} \rangle \ \mathcal{A}_{P,R} \ \langle \sigma_{\text{tgt}}, F_{\text{tgt}}, M_{\text{tgt}} \rangle \triangleq \\
((\sigma_{\text{tgt}} \neq \bot) \wedge \\
(\forall\, v_{\text{tgt}}.\ (\sigma_{\text{tgt}} = \textbf{return}(v_{\text{tgt}})) \Rightarrow \\
\quad \exists\, v_{\text{src}}.\ (\sigma_{\text{src}} = \textbf{return}(v_{\text{src}})) \ \wedge\ ((v_{\text{src}}, v_{\text{tgt}}) \in A) \ \wedge\ (F_{\text{tgt}} \cup R \sqsubseteq F_{\text{src}}) \ \wedge\ (M_{\text{tgt}} \sqsubseteq M_{\text{src}})) \wedge \\
(\forall\, \sigma'_{\text{tgt}}\ F'_{\text{tgt}}\ M'_{\text{tgt}}.\ (\langle \sigma_{\text{tgt}}, P, F_{\text{tgt}}, M_{\text{tgt}} \rangle \rightarrow \langle \sigma_{\text{tgt}}, P, F_{\text{tgt}}, M_{\text{tgt}} \rangle) \Rightarrow \\
\quad \exists\, \sigma'_{\text{src}}\ F'_{\text{src}}\ M'_{\text{src}}.(\langle \sigma_{\text{src}}, P, F_{\text{src}}, M_{\text{src}} \rangle \rightarrow^{*} \langle \sigma'_{\text{src}}, P, F'_{\text{src}}, M'_{\text{src}} \rangle) \ \wedge\ (\langle \sigma'_{\text{src}}, F'_{\text{src}}, M'_{\text{src}} \rangle \ \mathcal{A}_{P,R} \ \langle \sigma'_{\text{tgt}}, F'_{\text{tgt}}, M'_{\text{tgt}} \rangle)) \wedge \\
(\forall\, v\ \sigma'_{\text{tgt}}.\ (\sigma_{\text{tgt}} \xrightarrow{\texttt{choose}(v)} \sigma'_{\text{tgt}}) \Rightarrow \\
\quad \exists\, \sigma'_{\text{src}}.\ (\sigma_{\text{src}} \xrightarrow{\texttt{choose}(v)} \sigma'_{\text{src}}) \ \wedge\ (\langle \sigma'_{\text{src}}, F_{\text{src}}, M_{\text{src}} \rangle \ \mathcal{A}_{P,R} \ \langle \sigma'_{\text{tgt}}, F_{\text{tgt}}, M_{\text{tgt}} \rangle)) \wedge \\
(\forall\, e\ P'\ \sigma'_{\text{tgt}}\ F'_{\text{tgt}}\ M'_{\text{tgt}}.\ (\forall\, x\ v\ \sigma'_{\text{tgt}}.\ (\sigma_{\text{tgt}} \xrightarrow{\texttt{R}^{\texttt{rlx}}(x,v)} \sigma'_{\text{tgt}}) \Rightarrow \\
\quad \exists\, \sigma'_{\text{src}}.\ (\sigma_{\text{src}} \xrightarrow{\texttt{R}^{\texttt{rlx}}(x,v)} \sigma'_{\text{src}}) \ \wedge\ (\langle \sigma'_{\text{src}}, F_{\text{src}}, M_{\text{src}} \rangle \ \mathcal{A}_{P,R} \ \langle \sigma'_{\text{tgt}}, F_{\text{tgt}}, M_{\text{tgt}} \rangle)) \wedge \\
(\forall\, x\ v_{\text{tgt}}\ \sigma'_{\text{tgt}}.\ (\sigma_{\text{tgt}} \xrightarrow{\texttt{W}^{\texttt{rlx}}(x,v)} \sigma'_{\text{tgt}}) \Rightarrow \\
\quad \exists\, \sigma'_{\text{src}}\ v_{\text{src}}.\ (\sigma_{\text{src}} \xrightarrow{\texttt{W}^{\texttt{rlx}}(x,v)} \sigma'_{\text{src}}) \ \wedge\ (v_{\text{tgt}} \sqsubseteq v_{\text{src}}) \ \wedge\ (\langle \sigma'_{\text{src}}, F_{\text{src}}, M_{\text{src}} \rangle \ \mathcal{A}_{P,R} \ \langle \sigma'_{\text{tgt}}, F_{\text{tgt}}, M_{\text{tgt}} \rangle)) \wedge \\
(\forall\, x\ v\ \sigma'_{\text{tgt}}.\ (\sigma_{\text{tgt}} \xrightarrow{\texttt{R}^{\texttt{acq}}(x,v)} \sigma'_{\text{tgt}}) \Rightarrow \\
\quad \exists\, \sigma'_{\text{src}}.\ (\sigma_{\text{src}} \xrightarrow{\texttt{R}^{\texttt{acq}}(x,v)} \sigma'_{\text{src}}) \ \wedge\ (F_{\text{tgt}} \cup R \sqsubseteq F_{\text{src}}) \ \wedge\ \forall\, P'\ V.\ ((P \subseteq P') \ \wedge\ (dom(V) = P' \setminus P)) \ \Rightarrow \\
\qquad (\langle \sigma'_{\text{src}}, F_{\text{src}}, (\lambda x.(x \in dom(V))?V(x) : M_{\text{src}}(x)) \rangle \ \mathcal{A}_{P',\emptyset} \ \langle \sigma'_{\text{tgt}}, F_{\text{tgt}}, (\lambda x.(x \in dom(V))?V(x) : M_{\text{tgt}}(x)) \rangle)) \wedge \\
(\forall\, x\ v_{\text{tgt}}\ \sigma'_{\text{tgt}}.\ (\sigma_{\text{tgt}} \xrightarrow{\texttt{W}^{\texttt{rel}}(x,v_{\text{tgt}})} \sigma'_{\text{tgt}}) \Rightarrow \\
\quad \exists\, v_{\text{src}}\ \sigma'_{\text{src}}.\ (\sigma_{\text{src}} \xrightarrow{\texttt{W}^{\texttt{rel}}(x,v_{\text{src}})} \sigma'_{\text{src}}) \ \wedge\ (v_{\text{tgt}} \sqsubseteq v_{\text{src}}) \ \wedge\ \forall\, P'.\ (P \subseteq P') \ \Rightarrow \\
\qquad \exists R'.\ (F_{\text{tgt}} \cup R \subseteq F_{\text{src}} \cup R') \ \wedge\ (\forall x \notin R'.M_{\text{tgt}}(x) \sqsubseteq M_{\text{src}}(x)) \ \wedge\ (\langle \sigma'_{\text{src}}, \emptyset, M_{\text{src}} \rangle \ \mathcal{A}_{P',R'} \ \langle \sigma'_{\text{tgt}}, \emptyset, M_{\text{tgt}} \rangle)) \wedge \\
(\forall\, \Omega.\ \exists\, P'\ \sigma'_{\text{src}}\ F'_{\text{src}}\ M'_{\text{src}}\ tr. \\
\quad (\langle \sigma_{\text{src}}, P, F_{\text{src}}, M_{\text{src}} \rangle \xrightarrow{tr} \langle \sigma'_{\text{src}}, P', F'_{\text{src}}, M'_{\text{src}} \rangle) \ \wedge\ (tr \in \mathsf{Tr}(\Omega)) \ \wedge\ (\texttt{R}^{\texttt{acq}}(\_) \notin tr) \wedge \\
\quad ((\sigma'_{\text{src}} = \bot) \ \vee\ (F_{\text{tgt}} \cup R \subseteq F_{\text{src}} \cup \bigcup \{F \mid \texttt{W}^{\texttt{rel}}(\_,\_,\_,\_,F,\_) \in tr\}))) \vee \\
(\forall\, \Omega.\ \exists\, P'\ F'_{\text{src}}\ M'_{\text{src}}\ tr. \\
\quad (\langle \sigma_{\text{src}}, P, F_{\text{src}}, M_{\text{src}} \rangle \xrightarrow{tr} \langle \bot, P', F'_{\text{src}}, M'_{\text{src}} \rangle) \ \wedge\ (tr \in \mathsf{Tr}(\Omega)) \ \wedge\ (\texttt{R}^{\texttt{acq}}(\_) \notin tr))
\end{array}
$$

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

Figure 6. A simulation relation for SEQ

A The Simulation Relation in SEQ

The simulation relation in SEQ is given in Fig. 6. In order to prove adequacy of the simulation relation in Fig. 6, we define a simulation relation in PS^{na}, denoted by $\approx^{\text{na}}_{\text{PS}}$ between two threads in PS, and prove the adequacy.

Lemma A.1 (Adequacy of the simulation relation in PS^{na})**.** *If* $\sigma_{\text{src}} \approx^{\text{na}}_{\text{PS}} \sigma_{\text{tgt}}$*, then* $\sigma_{\text{tgt}} || \sigma_1 || \ldots || \sigma_n \sqsubseteq_{\text{PS}^{\text{na}}} \sigma_{\text{src}} || \sigma_1 || \ldots || \sigma_n$ *for any programs* $\sigma_1, \ldots, \sigma_n$*.*

Then, we prove that the simulation relation given in Fig. 6 implies the simulation relation in PS, $\approx_{\text{PS}}$.

Lemma A.2 (Simulation Lifting)**.** *If* $\sigma_{\text{src}} \sim_A \sigma_{\text{tgt}}$ *for a relation A, then* $\sigma_{\text{src}} \approx^{\text{na}}_{\text{PS}} \sigma_{\text{tgt}}$*.*

Therefore, we get the final adequacy theorem with respect to PS^{na}.

Theorem A.3 (Adequacy of the simulation relation in Fig. 6)**.** *If* $\sigma_{\text{src}} \sim_A \sigma_{\text{tgt}}$ *for a relation A, then* $\sigma_{\text{tgt}} || \sigma_1 || \ldots || \sigma_n \sqsubseteq_{\text{PS}^{\text{na}}} \sigma_{\text{src}} || \sigma_1 || \ldots || \sigma_n$ *for any programs* $\sigma_1, \ldots, \sigma_n$*.*

Theorem A.3 states the behavior refinement between two whole programs. However, we prove the congruence property, given in Fig. 7, which allows reasoning about 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.

$$\frac{A \text{ is reflexive}}{\sigma \sim_A \sigma}\ (\textsc{reflexivity}) \qquad \frac{A \subseteq A' \qquad \sigma_{\text{src}} \sim_A \sigma_{\text{tgt}}}{\sigma_{\text{src}} \sim_{A'} \sigma_{\text{tgt}}}\ (\textsc{monotonicity}) \qquad \frac{(v_{\text{src}}, v_{\text{tgt}}) \in A}{\mathbf{return}(v_{\text{src}}) \sim_A \mathbf{return}(v_{\text{tgt}})}\ (\textsc{return})$$

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

Figure 7. Compatibility Lemmas

B An Example Justifying the Semantics for Non-atomic writes

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:

$$\begin{array}{l} a := x^{\mathsf{na}}; \\ y^{\mathsf{rlx}} := a \end{array} \;\Big\|\; \begin{array}{l} b := y^{\mathsf{rlx}}; \\ c := \mathbf{freeze}(b); \\ \mathbf{if}\ c = 1\ \mathbf{then} \\ \quad x^{\mathsf{na}} := 1; \\ \quad \mathbf{print}(1) \\ \mathbf{else} \\ \quad x^{\mathsf{na}} := 2 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} b := y^{\mathsf{rlx}}; \\ c := \mathbf{freeze}(b); \\ x^{\mathsf{na}} := 2; \\ \mathbf{if}\ c = 1\ \mathbf{then} \\ \quad x^{\mathsf{na}} := 1; \\ \quad \mathbf{print}(1)\ \ \textit{// reachable!} \end{array}$$

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 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[11] 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 $x^{\mathsf{na}} := 1$. (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 $x^{\mathsf{na}} := 1$.) 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.

C A Problem of PS with Non-determinism

In this section, we observe that PS (as well as PS2 and PS2.1) does not validate reordering of internal (pure) non-determinism[12] followed by a release write. The source of the problem is that release writes explicitly block promises with non-$\bot$ 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-$\bot$ message view to x.) Consider the following program:

$$\begin{array}{l} a := x^{\mathsf{rlx}}; \\ y^{\mathsf{rlx}} := a \end{array} \;\Big\|\; \begin{array}{l} b := \mathbf{freeze}(\mathsf{undef}); \\ x^{\mathsf{rel}} := 0; \\ \mathbf{if}\ b = 1\ \mathbf{then} \\ \quad c := y^{\mathsf{rlx}}; \\ \quad \mathbf{if}\ c = 1\ \mathbf{then} \\ \quad\quad x^{\mathsf{rlx}} := 1; \\ \quad\quad \mathbf{print}(1)\ \ \textit{// not reachable!} \\ \mathbf{else} \\ \quad x^{\mathsf{rlx}} := 1 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} x^{\mathsf{rel}} := 0; \\ b := \mathbf{freeze}(\mathsf{undef}); \\ \mathbf{if}\ b = 1\ \mathbf{then} \\ \quad c := y^{\mathsf{rlx}}; \\ \quad \mathbf{if}\ c = 1\ \mathbf{then} \\ \quad\quad x^{\mathsf{rlx}} := 1; \\ \quad\quad \mathbf{print}(1)\ \ \textit{// reachable!} \\ \mathbf{else} \\ \quad x^{\mathsf{rlx}} := 1 \end{array}$$

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.

[11] 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.
[12] a representative example of such non-determinism is **freeze** instruction of LLVM IR [21].

Domain: $D \in \mathsf{Loc} \to \mathcal{P}(\mathsf{Reg})$
Ordering: $D_1 \sqsubseteq D_2 \triangleq \forall x.\ D_1(x) \supseteq D_2(x)$
Transitions:

$$
\begin{array}{lll}
T(x)(x^{\mathsf{na}} := v, t) & = \emptyset & \\
T(x)(a := x^{\mathsf{na}}, t) & = t \cup \{a\} & \\
T(x)(\_ := y^{\mathsf{acq}}, t) & = \emptyset & \\
T(x)(\_, t) & = t & \text{otherwise}
\end{array}
$$

(a) Analysis for Load-to-Load Forwarding

Domain: $D \in \mathsf{Loc} \to \{\circ, \bullet, \top\}$
Ordering: $\circ \sqsubseteq \bullet \sqsubseteq \top$
Transitions:

$$
\begin{array}{lll}
T_B(x)(x^{\mathsf{na}} := \_, t) & = \circ & \\
T_B(x)(\_ := x^{\mathsf{na}}, t) & = \top & \\
T_B(x)(y^{\mathsf{rel}} := \_, \bullet) & = \top & \\
T_B(x)(\_ := y^{\mathsf{acq}}, \circ) & = \bullet & \\
T_B(x)(\_, t) & = t & \text{otherwise}
\end{array}
$$

(b) Analysis for Dead Store Elimination

Figure 8. Analyses for optimizer passes

First, π_2 can print 1 through following execution:

(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.)

D A Certified Optimizer

In this section, we describe analyses and optimizations 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. 8a and Fig. 8b 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 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 $a := x^{\mathsf{na}}$; 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 $a := x^{\mathsf{na}}$ 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. 8b. 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 $x^{\mathsf{na}} := \_$ 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 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.

E A challenge in supporting mixing of atomic and non-atomic accesses to the same location

We describe a challenge in establishing the adequacy theorem of SEQ (*i.e.*, Theorem 6.2) under the presence of mixing of atomic and non-atomic accesses to the same location. The problem stems from a common compiler transformation that replaces an `undef` in the source program with a non-`undef` value. In order to validate this transformation, PS$^{\mathsf{na}}$ 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:

$$\begin{array}{l} c := y^{\mathsf{rlx}}; \\ \textbf{if } c = 1 \textbf{ then} \\ \quad x^{\mathsf{rlx}} := 1 \\ \textbf{else} \\ \quad x^{\mathsf{rlx}} := \mathsf{undef} \end{array} \rightsquigarrow \begin{array}{l} c := y^{\mathsf{rlx}}; \\ \textbf{if } c = 1 \textbf{ then} \\ \quad x^{\mathsf{rlx}} := 1 \\ \textbf{else} \\ \quad x^{\mathsf{rlx}} := \mathsf{undef} \end{array} \rightsquigarrow \begin{array}{l} c := y^{\mathsf{rlx}}; \\ \textbf{if } c = 1 \textbf{ then} \\ \quad x^{\mathsf{rlx}} := 1 \\ \textbf{else} \\ \quad x^{\mathsf{rlx}} := 1 \end{array} \rightsquigarrow \begin{array}{l} x^{\mathsf{rlx}} := 1; \\ c := y^{\mathsf{rlx}} \end{array}$$

After the second transformation, which is marked in red, the thread can promise $x = 1$ before executing $c := y^{\mathsf{rlx}}$ 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 $x^{\mathsf{rlx}} := \mathsf{undef}$ cannot fulfill the promise $x = 1$. The lower operation solves this problem by allowing the thread to first lower its promise $x = 1$ to $x = \mathsf{undef}$ and then fulfill the promise through the write $x^{\mathsf{rlx}} := \mathsf{undef}$.

While the lowering admits above transformation, it causes a mismatch between SEQ and PS$^{\mathsf{na}}$. In particular, PS$^{\mathsf{na}}$ allows a value of existing message that is pointed by a thread's view to be lowered by another thread. 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 PS$^{\mathsf{na}}$ invalidates the proof sketch for the adequacy given in §6.

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.