Verifying Optimizations of Concurrent Programs in the Promising Semantics

Junpeng Zha
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
jpzha@smail.nju.edu.cn

Hongjin Liang*
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
hongjin@nju.edu.cn

Xinyu Feng
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
xyfeng@nju.edu.cn

Abstract

Weak memory models for concurrent programming languages are expected to admit standard compiler optimizations. However, prior works on verifying optimizations in weak memory models are mostly focused on simple optimizations on small code snippets which satisfy certain syntactic requirements. It receives less attention whether weak memory models can admit real-world optimization algorithms based on program analyses.

In this paper, we develop the first simulation technique for verifying thread-local analyses-based optimizations in the promising semantics PS2.1, which is a weak memory model recently proposed for C/C++11 concurrency. Our simulation is based on a novel non-preemptive semantics, which is equivalent to the original PS2.1 but has less non-determinism. We apply our simulation to verify four optimizations in PS2.1: constant propagation, dead code elimination, common subexpression elimination and loop invariant code motion.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program analysis**; • **Software and its engineering** → **Formal software verification**; *Compilers*; *Concurrent programming languages.*

Keywords: Code optimization, Weak memory models, Concurrency, Verification, Simulations

ACM Reference Format:
Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. 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, 15 pages. https://doi.org/10.1145/3519939.3523734

*Corresponding author.

1 Introduction

The design of weak memory models of concurrent programming languages should balance the conflicting demands of hardware, programmers, and compilers. The promising semantics (PS) [10, 13] is recently proposed for trying to satisfy all the demands as an operational weak memory model for C11-like languages. It has been shown that PS is implementable for mainstream hardware platforms (e.g. x86-TSO and Power), and also friendly for programmers as it provides DRF guarantees and avoids the bad "out-of-thin-air" behaviors. In this paper, we focus on the demands of compilers: does PS admit compiler optimizations, especially those already used heavily in existing compilers?

In particular, we are curious about whether PS allows *thread-local analyses-based optimizations*. They are commonly found in mainstream C compilers (e.g. GCC and LLVM), but rarely studied in existing works on weak memory models. Most existing works only study whether it is sound to do optimizations on code snippets (e.g. [3, 16, 22, 23]). These optimizations can eliminate redundant reads and writes and reorder independent instructions, but usually require the source code snippet to conform to a certain syntactic "shape", e.g. the first write can be eliminated for two adjacent writes to the same location in the same access mode. By contrast, analyses-based optimizations are more complex, since their transformations are based on the results of program analysis algorithms, which usually depend on program behaviors. For instance, dead code elimination (DCE) eliminates a write to a location if the value of that location is not used later in any execution of the program. It can eliminate not only redundant writes, but also those that are found dead in semantics. For PS, people have shown that it admits thread-local optimizations on code snippets [10] and some global optimizations such as register promotion [13], but what about thread-local analyses-based optimizations such as DCE?

Unfortunately the answer is not obvious at all. Consider the optimization of loop invariant code motion (LICM) as an example, which will move loop invariants (e.g. a read of a variable that gives the same value in every iteration) out of the loop body. A naive adaption of LICM to PS can transform `foo()` to `foo_opt()` in Fig. 1, by ignoring the subscripts (e.g. acq and na) annotated for weak semantics. We call `foo()` and

```
1  int foo() {                     1′  int foo_opt() {
2    int r1 := 0, r2 := 0;         2′    int r1 := 0, r2 := 0;
3    while(r1 < 10) {              3′    r2 := y_na;
4      while(x_acq == 0);          4′    while(r1 < 10) {
5      r2 := y_na;                 5′      while(x_acq == 0);
6      r1 := r1 + 1;               6′      r1 := r1 + 1;
7    }                             7′    }
8    return r2;                    8′    return r2;
9  }                               9′  }
```

Figure 1. Example of loop invariant code motion.

`foo_opt()` the *source* and the *target* of the transformation respectively. Here `x` and `y` are global variables. The read of `y` inside the loop body in the source code (line 5) is moved out of the loop in the target (line 3′).

Soundness of a transformation is defined by the *refinement*: the target does not produce more behaviors than the source. We find that Fig. 1 is *unsound* in PS (and in the standard C/C++11 model). To see why, suppose `x` and `y` are initialized to 0, and another thread runs `g()` in parallel:

$$\texttt{void g()\{ } \texttt{y}_{\texttt{na}} := 1; \texttt{x}_{\texttt{rel}} := 1; \}$$

Then r_2 in `foo_opt()` may see `y`'s initial value 0 as well as the new value 1, but r_2 in `foo()` can only read 1 due to the release-acquire synchronization. Thus `foo_opt()` ∥ `g()` does not refine `foo()` ∥ `g()` in PS.

That said, we don't want to conclude from Fig. 1 that LICM is unsound in PS. Instead, we want to verify that, with careful adaptions, LICM and other well-known sequential optimizations are all sound in PS. For Fig. 1, the transformation will become sound if we change the acquire reads to relaxed reads (at line 4 in `foo()` and line 5′ in `foo_opt()`). Consequently, we are more convinced that PS does admit LICM, where loop invariants can be moved around carefully chosen atomic accesses. The only question is, how to formally verify it.

Note that as we expect PS to admit *optimization passes* or *optimization algorithms* (such as LICM), it is insufficient if we only show the correctness of transformation of particular programs. Instead we should prove the transformation correctness (i.e. refinement) on *arbitrary* reasonable programs.

For this, the standard proof technique is to build simulations. The simulation should be *compositional* (a.k.a. congruent) in that it is preserved by the language constructs such as sequential composition, branches, loops and parallel composition. Then, by induction on the program structure of the source, we can prove simulations for all pairs of source and target programs. Since simulation ensures refinement, we can conclude the correctness of an optimization.

Simulations (including the one in CompCert) can easily ensure compositionality w.r.t. sequential language constructs. So, the main challenge we face is, how to design a *thread-local* simulation that has *both* parallel compositionality (a.k.a. horizontal compositionality) in PS *and* applicability for various optimizations. Our starting point is the simulation *with an invariant parameter* [8, 14, 21], a key proof technique to achieve parallel compositionality in the sequentially consistent (SC) semantics. The invariant parameter provides an abstraction for the interference between the current thread and others, but for PS, it may have to expose many complex details of the weak semantics. Consequently, verifiers may find it quite hard to instantiate the invariant. To simplify the invariant, our ideas are, 1) introducing a non-preemptive semantics, to reduce the program points where interference occurs; and 2) assuming that source programs are free of write-write races, to reduce what interference can occur.

In this paper, we present the first simulation technique which is parallel compositional and applicable for verifying thread-local analyses-based optimizations in PS2.1 [5, 13], the up-to-date version of the promising semantics. We support *all* the language features supported by PS2.1, which are all features of C11 concurrency except consume reads and SC accesses. Also, we focus on optimizations on *non-atomic* accesses (i.e. those annotated with na). These optimizations modify non-atomic accesses only, but it is possible to move non-atomic accesses around atomic ones (like Fig. 1) or use knowledge from other parts of code across atomic accesses. We do not consider optimizations on atomic accesses, as they are rarely found in mainstream C compilers such as GCC and LLVM. We make the following new contributions:

First, we show that PS2.1 is equivalent to a *non-preemptive semantics* where context switch and promise/reserve steps are forbidden inside the execution of code blocks consisting of non-atomic accesses only. The non-preemptive semantics is interesting in its own right, since it reduces non-determinism, making it potentially easier to reason about program behaviors in the promising semantics.

Second, we formulate write-write race freedom in PS2.1. Intuitively a *write-write race* means that two threads both (non-atomically) write to the same location, and neither write happens before the other. Assuming write-write race freedom of source programs allows verifiers to easily instantiate the invariant in simulations, without concerning much about the details of PS (e.g. how promises are certified). Note that we allow the source to have read-write races, as some sound optimizations (e.g. LICM) do introduce read-write races.

Third, based on the non-preemptive semantics, we propose a thread-local simulation for verifying optimizations. Our simulation is compositional w.r.t. parallel composition, as long as the source program is free of write-write races. It is parameterized with an invariant specifying the possible interference from environment (i.e. other threads). In compliance with the non-preemptive semantics, the environment interference occurs only outside of the current thread's execution of non-atomic accesses, which makes it easy to verify transformations that reorder non-atomic accesses.

Fourth, we have applied our simulation to verify four optimization algorithms, including constant propagation (ConstProp), dead code elimination (DCE), common subexpression elimination (CSE) and loop invariant code motion (LICM).

The four optimizations are, for the first time, adapted to PS2.1. In particular, we carefully allow optimizations across certain atomic accesses, e.g. LICM is allowed across a relaxed read/write or a release write, but not an acquire read; and DCE is allowed across a relaxed read/write or an acquire read, but not a release write.

Finally, we prove the adequacy of our verification method (i.e. the simulation), saying that it ensures optimization correctness for write-write race-free source programs. We have mechanized the proofs of the verification framework in the Coq proof assistant. The Coq mechanization of the proofs of the optimization algorithms is still in progress.

In the rest of this paper, we first explain our main ideas for verifying optimizations in PS in Sec. 2, and review the promising semantics in Sec. 3. Then we present the equivalent non-preemptive semantics in Sec. 4, and formulate write-write race freedom in Sec. 5. We propose our thread-local simulation and the final theorem in Sec. 6, introduce the verified optimizations in Sec. 7, and discuss related work in Sec. 8. Supplementary materials for this paper, including the appendix and the Coq development, are available at [25].

2 Informal Development

We quickly review, in Sec. 2.1, the main ideas of PS; and in Sec. 2.2, the idea of parameterizing simulation with invariant to ensure parallel compositionality in the SC semantics. Then we introduce the key challenges and our basic ideas in developing simulations for verifying optimizations in PS. In Sec. 2.3 and 2.4, we motivate the non-preemptive semantics and write-write race freedom, respectively. We discuss why we allow read-write races in Sec. 2.5, and give an overview of our verification framework in Sec. 2.6.

2.1 The Promising Semantics: An Overview

The promising semantics (PS) introduces several key ideas to model the out of order execution in C11-like memory models. It keeps the whole history of memory updates by recording all writes as time-stamped messages in the memory, so that a read may see more than one prior writes. In particular, a read needs not read the "latest" write. This allows the following annotated outcome for the store buffering example (SB). (Throughout the paper, we call the left thread t_1 and the right t_2, and assume all the variables are initialized to 0.)

$$\begin{array}{l|l} x_{\mathsf{rlx}} := 1; & y_{\mathsf{rlx}} := 1; \\ r_1 := y_{\mathsf{rlx}};\ \textit{// 0} & r_2 := x_{\mathsf{rlx}};\ \textit{// 0} \end{array} \qquad \text{(SB)}$$

PS also allows a thread to *promise* a write at any time without actually executing the write command. Like regular writes, the promised writes can be read by other threads. This allows the load buffering example (LB):

$$\begin{array}{l|l} r_1 := x_{\mathsf{rlx}};\ \textit{// 1} & r_2 := y_{\mathsf{rlx}};\ \textit{// 1} \\ y_{\mathsf{rlx}} := 1; & x_{\mathsf{rlx}} := r_2; \end{array} \qquad \text{(LB)}$$

The following execution gives us the annotated outcome. We let t_1 first promise to write y, and let t_2 read from the promised write. At the end t_1 fulfills its promise by executing the actual write instruction.

$$[t_1 : \text{promise}\ (y_{\mathsf{rlx}} := 1);\ t_2 :\ r_2 := y_{\mathsf{rlx}}\ \textit{//1};\ t_2 :\ x_{\mathsf{rlx}} := r_2;\\ t_1 :\ r_1 := x_{\mathsf{rlx}}\ \textit{//1};\ t_1 :\ y_{\mathsf{rlx}} := 1\ (\text{fulfill})].$$

A thread can only promise to write if it can *thread-locally certify* that its promise will be fulfilled by itself. This avoids the "out-of-thin-air" reads. For (LB), if we change t_1's write to $y_{\mathsf{rlx}} := r_1$, the outcome 1 would be "out-of-thin-air". It is disallowed in PS, because t_1 cannot promise $y_{\mathsf{rlx}} := 1$, as it is not able to fulfill the promise when running in isolation.

Moreover, PS requires the promise to be certified at the *capped memory*, a special extension of the current memory. Certifying promises only from the current memory is insufficient, because it completely ignores the possible interference by other threads, which could make the current thread unable to fulfill its promises. In particular, when two threads t and t′ perform compare-and-swap (CAS) operations reading from the same write, the current thread t should not make a promise by assuming that its CAS will succeed, since t′ may succeed first in the actual execution. The construction of the capped memory models the environment interference (e.g. the successful CAS performed by t′). More technically, it reserves all the timestamps falling between the timestamps of existing writes, together with the timestamp next to the greatest assigned one (i.e. a *cap*), so that the current thread can no longer assign them to its future writes.

We will review PS in more detail in Sec. 3.

2.2 Simulation and Parallel Compositionality

We write a concurrent source program $\mathbb{P}_s$ in the form of **let** π_s **in** $f_1 \parallel \ldots \parallel f_n$, consisting of declarations π_s for a set of functions and n threads calling the functions $f_1, \ldots, f_n$ in π_s. An optimization pass Opt transforms π_s to the target π_t. So the target program $\mathbb{P}_t$ is **let** π_t **in** $f_1 \parallel \ldots \parallel f_n$. In general, verifying Opt requires one to verify the refinement $\mathbb{P}_s \supseteq \mathbb{P}_t$, saying that every observable event trace generated by the execution of $\mathbb{P}_t$ can also be generated by the execution of $\mathbb{P}_s$.

Simulation for refinement proofs. To prove $\mathbb{P}_s \supseteq \mathbb{P}_t$, the standard approach is to construct an (upward) simulation relation $\mathbb{P}_s \succcurlyeq \mathbb{P}_t$, depicted in Fig. 2(a). It requires that any step of $\mathbb{P}_t$ should correspond to zero-or-more steps of $\mathbb{P}_s$ such that the simulation $\succcurlyeq$ still holds between the resulting source and target programs $\mathbb{P}'_s$ and $\mathbb{P}'_t$.

This approach is used in CompCert for verifying compilation of sequential programs. But the simulation relates whole

$$\begin{tikzcd} \mathbb{P}_s \arrow[r, "*"] \arrow[d, dashed, no head, "\succcurlyeq"'] & \mathbb{P}'_s \arrow[d, dashed, no head, "\succcurlyeq"] \\ \mathbb{P}_t \arrow[r] & \mathbb{P}'_t \end{tikzcd}$$

(a) $\mathbb{P}_s \succcurlyeq \mathbb{P}_t$

$$\begin{tikzcd} \circ \arrow[r, "*"] \arrow[d, dashed, no head, "\succcurlyeq"'] & \circ \arrow[r, thick] \arrow[d, dashed, no head, "\succcurlyeq"', "I"] & \circ \arrow[r, "*"] \arrow[d, dashed, no head, "I"', "\succcurlyeq"] & \circ \arrow[d, dashed, no head, "\succcurlyeq"] \\ \circ \arrow[r] & \circ \arrow[r, thick] & \circ \arrow[r] & \circ \end{tikzcd}$$

(b) $I \vdash (\pi_s, f) \succcurlyeq (\pi_t, f)$

Figure 2. Simulations, without and within environment.

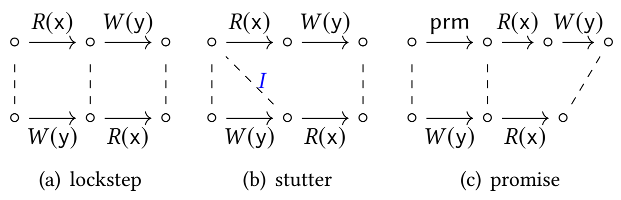

Figure 3. Simulations for (Reorder). We write $R(\mathsf{x})$, $W(\mathsf{y})$ and prm for $r := \mathsf{x}_{\mathsf{na}}$, $\mathsf{y}_{\mathsf{na}} := 2$ and promise, respectively. The target execution is at the bottom, while source is at the top.

programs only, and it is *not* parallel compositional. That is, one cannot derive $\mathbb{P}_s \succcurlyeq \mathbb{P}_t$ from $\forall i.$ **let** π_s **in** $\mathsf{f}_i \succcurlyeq$ **let** π_t **in** f_i (i.e. the simulation between individual threads). The reason is that the simulation does not take into account the interactions with environment (i.e. the other threads) which may update the shared resource.

Invariant for parallel compositionality. Prior works have developed thread-local simulations which are parallel compositional *in the SC semantics* [8, 14, 21]. The simulations are established for individual threads, without relying on the code of other threads. The behaviors of the other threads are abstracted as a *rely* condition, an invariant over environment state transitions. As shown in Fig. 2(b), the thread-local simulation $I \vdash (\pi_s, \mathsf{f}) \succcurlyeq (\pi_t, \mathsf{f})$ is parameterized with the invariant I relating source and target shared states, which is required to hold at every switch point. That is, the current thread's transitions (the thin arrows) must ensure I when switching out, and the environment transitions (the thick arrows) must give back I when switching back. Such a simulation is compositional in the SC semantics: one can derive $\mathbb{P}_s \succcurlyeq \mathbb{P}_t$ from $\forall i.\ I \vdash (\pi_s, \mathsf{f}_i) \succcurlyeq (\pi_t, \mathsf{f}_i)$.

However, this invariant approach does not directly extend to PS. For the approach to work, since I specifies the interference at switch points, one needs to first figure out where the switch points are and what the interference is. In PS, the answers to these two questions are not obvious, as apparent answers would make it hard to instantiate I.

2.3 The Need of Non-Preemptive Semantics

Let's consider the first question: *where is a switch point?* In PS, threads execute in an interleaved fashion, so it seems natural to treat every program point as a possible switch point. Consequently, the invariant I in the simulation has to be weak enough to hold at every step.

For example, consider how to build the simulation for the instruction reordering transformation (Reorder):

$$r := \mathsf{x}_{\mathsf{na}}; \mathsf{y}_{\mathsf{na}} := 2; \quad \rightsquigarrow \quad \mathsf{y}_{\mathsf{na}} := 2; r := \mathsf{x}_{\mathsf{na}}; \qquad \text{(Reorder)}$$

Figure 3 lists three typical ways to relate the source and target executions. However, none of them is good enough.

In Fig. 3(a), we let the first target step correspond to the read step of the source. Then we will have to decide which value the source should read, before the same read is performed at the target. On the one hand, the simulation requires that the read values be the same at the two sides. On the other hand, we cannot predict the read value of a future target step since the read value can be non-deterministic in PS. As such, we are unable to pick a "correct" read value for the source, so we cannot build the simulation in Fig. 3(a). Similarly, it is also infeasible if we let the first target step correspond to the source executing both $R(\mathsf{x})$ and $W(\mathsf{y})$.

In Fig. 3(b), the first target step $W(\mathsf{y})$ corresponds to zero source step. Since the point after the first target step is a switch point, the invariant I should hold (as explicitly labeled in the figure). So, I has to allow the target memory to contain more writes of y than the source. Considering the environment interference which may possibly write to x, it seems that I needs to be weaker, allowing the target memory to also contain more writes of x! Consequently, the next target step $R(\mathsf{x})$ may read a value not possible by the source, breaking this simulation.

One may think that the break of Fig. 3(b) is caused by races, as we allow the environment to write x when the current thread reads x. However, (Reorder) is actually sound in PS for any programs including racy ones, because we can construct the simulation in Fig. 3(c). For the first target step $W(\mathsf{y})$, we let the source thread promise to write y. Then we can have a strong I saying that the target and source memories are the same. Then the next $R(\mathsf{x})$ steps at the two sides can read the same value. The simulation in Fig. 3(c) works well for the simple example (Reorder), but it is not easy to use for more general reordering transformations such as

$$r := \mathsf{x}_{\mathsf{na}}; C; \mathsf{y}_{\mathsf{na}} := 2; \rightsquigarrow \mathsf{y}_{\mathsf{na}} := 2; C; r := \mathsf{x}_{\mathsf{na}};$$

Here the code C does not involve x, y or r, but it can be arbitrarily complex, making it non-trivial to certify the promise of the source thread.

Non-preemptive semantics to simplify invariant. Let us step back to the simulation in Fig. 3(b). We want to argue that its key problem is that too many switch points make I too weak. To address the problem, we build simulations over non-preemptive semantics which has much less switch points than the interleaving semantics. In particular, non-preemptive semantics forbids context switch inside the execution of code blocks consisting of only non-atomic accesses. Thus, for (Reorder), the program point after the first target step is not a switch point in the non-preemptive semantics. As such, we can use the simple I (the same one for Fig. 3(c)) saying that the source and target memories are always the same at every switch point (in this example, the only switch points are before and after the whole code segment).

The remaining question is, can we really verify refinement in interleaving semantics by building simulations in non-preemptive semantics? The prior work CASCompCert [8] shows that this proof path is sound *for SC semantics and for data-race-free programs*, because for them the interleaving

```
1  r1 := y_rlx;                    ||  4  r2 := x_rlx;
2  if (r1 == 1) z_na := 1;         ||  5  if (r2 == 1) {
3  else x_rlx := 1;                ||  6      z_na := 2; y_rlx := 1; }
```

Figure 4. Does the program have a write-write race?

semantics is equivalent to the non-preemptive semantics. In this work, we adopt similar ideas in the setting of PS2.1. We design a non-preemptive version for PS2.1 and prove that the equivalence between the two semantics *unconditionally holds for any programs.* We will explain the details in Sec. 4.

2.4 The Need of Write-Write Race Freedom

Let's consider the second question: *what is the interference?* In PS, there are two sources of interference: one is the behaviors of the other threads running in parallel, as in any concurrency semantics; and the other is the construction of the capped memory in the current thread's promise certification, as explained in Sec. 2.1. Asking the invariant I to specify both of them would make it difficult to define and use, because the construction of the capped memory can be different from the *actual* behaviors of the other threads. In particular, to establish the simulation, one can choose a specific execution strategy for a source thread to correspond to the target. If we know all the source threads would follow the same strategy, their behaviors may follow certain pattern instead of being arbitrary, which can be encoded into I and simplify the simulation proof. However, I has to be much weaker (and potentially more complicated) if it also needs to cover the construction of the capped memory as another source of interference.

To address this problem, we do not require I to cover the construction of the capped memory. Instead, we only consider the correctness of optimizations for source programs that are free of write-write races, based on the observation that a thread from a write-write-race-free program can certify promises (for non-atomic writes) against the current memory instead of the capped memory.

Intuitively a *write-write race* means that two threads both (non-atomically) write to the same location, and neither write happens before the other. So, write-write race freedom forbids a thread t to write to a location when the memory contains a write of the same location made by another thread t′ and unobserved by t. This gives the same technical effect as the capped memory: t cannot write a message m when the memory already contains another message m' at the same location with a higher timestamp written by t′.

Subtleties in formulating write-write race freedom. One needs to be careful with the promises when defining write-write race freedom in PS. For example, one may think that the program in Fig. 4 has the following execution:

$$[\mathtt{t}_1: \text{promise } (\mathtt{x}_{\mathtt{rlx}} := 1);\ \mathtt{t}_2: r_2 := \mathtt{x}_{\mathtt{rlx}} // 1;\ \mathtt{t}_2: \mathtt{z}_{\mathtt{na}} := 2;$$
$$\mathtt{t}_2: \mathtt{y}_{\mathtt{rlx}} := 1;\ \mathtt{t}_1: r_1 := \mathtt{y}_{\mathtt{rlx}} // 1; \mathtt{t}_1: \mathtt{z}_{\mathtt{na}} := 1 // \text{Race?}].$$

```
                          r := x_na;                  r := x_na;
while(r1 < 8){            while(r1 < 8){              while(r1 < 8){
  r2 := x_na;     LInv      r2 := x_na;      CSE        r2 := r;
  r1 := r1 + 1;    ~>       r1 := r1 + 1;     ~>        r1 := r1 + 1;
}                         }                           }
(denoted by C_src)        (denoted by C_m)            (denoted by C_tgt)
```

(a) LICM first performs LInv, and then CSE

```
r0 := y_acq;                  r0 := y_acq;              g() {
if(r0 == 1) {         LInv    if(r0 == 1) {               z_na := 9;
    r1 := z_na;        ~>         r1 := z_na;             y_rel := 1;
    C_src;                        C_m;                    x_na := 5;
}                             }                         }
```

(b) LInv may introduce read-write races

Figure 5. More examples of LICM.

From this execution, one would think that a write-write race on z occurs, because $\mathtt{t}_1$ and $\mathtt{t}_2$'s writes to z are not synchronized. However, after $\mathtt{t}_1$ reads 1 from y at line 1, its earlier promise would never be fulfilled, so this execution should not be considered legal.

It seems more natural to not view that this program has a write-write race. Here $\mathtt{t}_1$ writes to z only when r_1 sees 1; while $\mathtt{t}_2$ writes to z only when r_2 obtains 1, which means r_1 of $\mathtt{t}_1$ must see 0. As such, the two threads never write to z in the same execution, so there is no write-write race.

To reflect the intuition, we carefully define write-write race freedom by *checking races only when promises are certified.* By our definition (presented in Sec. 5), the program in Fig. 4 does not have a write-write race.

2.5 Allowing Read-Write Races

Unlike write-write races, we *allow* the source programs to have read-write races. This is crucial for supporting optimizations performing redundant read introduction, which can introduce read-write races.

A typical example is loop invariant code motion (LICM). It is implemented by composing two optimization passes: LICM $\triangleq$ (LInv $\circ$ CSE), where $\circ$ denotes the composition of two optimizers (called "vertical composition"). The first pass LInv detects the loop invariant and allocates a fresh register at the entry of the loop to save the value of the loop invariant. This pass can introduce redundant memory reads. The second pass is common subexpression elimination (CSE) which eliminates the evaluation of loop invariant inside the loop body. In Fig. 5(a), in the source code C_{src}, x is a loop invariant. LInv *introduces a redundant read* $r := \mathtt{x}_{\mathtt{na}}$ before the loop. Next, CSE keeps $r := \mathtt{x}_{\mathtt{na}}$ and replaces the read of x in the loop body with a read of r. So in the target C_{tgt}, we have moved the read of x out of the loop.

LInv, the first pass of LICM, may *introduce read-write races*, as shown in the example of Fig. 5(b). Here the code snippets C_{src} and C_m in the source and target code are those presented in Fig. 5(a). There is no race if one thread $\mathtt{t}_1$ runs the source code and another thread $\mathtt{t}_2$ runs g() in parallel, because the

$$\begin{tikzcd}
\mathbb{P}_s \supseteq \mathbb{P}_t \arrow[d, phantom, "⑤\ \rotatebox[origin=c]{90}{$\subseteq$} \qquad \rotatebox[origin=c]{-90}{$\subseteq$}\ ⑤"] & & & \text{ww-RF}(\mathbb{P}_t) \\
\hat{\mathbb{P}}_s \supseteq \hat{\mathbb{P}}_t & & & \\
\boxed{\hat{\mathbb{P}}_s \succcurlyeq \hat{\mathbb{P}}_t} \arrow[u, Rightarrow, "④"] & \wedge & & \text{ww-NPRF}(\hat{\mathbb{P}}_t) \arrow[u, Rightarrow, "①"'] \\
 & \arrow[u, Rightarrow, "③"] & & \text{ww-NPRF}(\hat{\mathbb{P}}_s) \arrow[ll, Rightarrow] \\
 & \boxed{I \vdash (\pi_s, \mathsf{f}) \succcurlyeq (\pi_t, \mathsf{f}),\ \text{for any } \mathsf{f}} \arrow[u, Rightarrow, no head] & & \\
 & \boxed{\text{Verif(Opt)} \wedge \text{Opt}(\pi_s) = \pi_t} \arrow[u, Rightarrow, "②"] & & \text{ww-RF}(\mathbb{P}_s) \arrow[uu, Rightarrow, "①"']
\end{tikzcd}$$

Figure 6. Our proof path. Here $\mathbb{P}_s$ = (**let** π_s **in** $\mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_n$), $\mathbb{P}_t$ = (**let** π_t **in** $\mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_n$), $\hat{\mathbb{P}}_s$ = (**let** π_s **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$) and $\hat{\mathbb{P}}_t$ = (**let** π_t **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$).

acquire-release synchronization on y ensures that r_1 must read 9, so t_1 does not enter the loop to access x. But, if t_1 runs the target code, since x is read outside of the loop in C_m, there is a race between t_1's read and t_2's write of x. Since the resulting code with read-write races might become the input source code for subsequent optimization passes (e.g. `CSE`, in the case of `LICM`), we decide to not assume the absence of read-write races in the source.

Note that introduction of *redundant* reads is sound in PS, even in the presence of read-write races. Although duplicated reads may see different values with read-write races, this is fine since only one of the read values is used (i.e. other reads are redundant). We have verified `LInv` in PS2.1 (in fact, our proof for `LICM` is composed from those for `LInv` and `CSE`).

2.6 Our Proof Path

Figure 6 shows our proof path for the adequacy of our verification method, i.e. how we reduce the problem of verifying optimization passes to verifying thread-local simulations in non-preemptive semantics.

As we explained, verifying an optimization pass Opt requires us to verify the refinement $\mathbb{P}_s \supseteq \mathbb{P}_t$ in PS2.1, for any source and target pairs $\mathbb{P}_s$ and $\mathbb{P}_t$, as long as $\mathbb{P}_s$ is free of write-write races (i.e. ww-RF($\mathbb{P}_s$) holds). In Fig. 6, we reduce the goal $\mathbb{P}_s \supseteq \mathbb{P}_t$ to $\hat{\mathbb{P}}_s \supseteq \hat{\mathbb{P}}_t$, where $\hat{\mathbb{P}}$ denotes the program in our non-preemptive semantics, using the equivalence between our non-preemptive semantics and PS2.1 (see ⑤). We also show that ww-RF($\mathbb{P}$) is equivalent to ww-NPRF($\hat{\mathbb{P}}$) (see ①) where ww-NPRF is the counterpart of ww-RF in the non-preemptive semantics. The correctness Verif(Opt) ensures the thread-local simulation, $I \vdash (\pi_s, \mathsf{f}) \succcurlyeq (\pi_t, \mathsf{f})$, for any function f (see ②). Here the invariant I can be instantiated differently when verifying different optimizations. The simulation is compositional (see ③) and the resulting whole-program simulation $\hat{\mathbb{P}}_s \succcurlyeq \hat{\mathbb{P}}_t$ ensures the refinement in our non-preemptive semantics (see ④). Our verification method

$$
\begin{array}{rrcl}
\textit{(Lab)} & \mathsf{f} & \in & \mathbb{N} \qquad\qquad \textit{(Val)}\ v\ \in\ \text{Int32} \\
\textit{(Reg)} & r & ::= & \ldots \quad \textit{(Var)}\ \mathsf{x},\mathsf{y},\mathsf{z} ::= \ldots \quad \textit{(Atms)}\ \iota \subseteq \textit{Var} \\
\textit{(ModeR)} & o_r & ::= & \text{na} \mid \text{rlx} \mid \text{acq} \quad \textit{(ModeW)}\ o_w\ ::=\ \text{na} \mid \text{rlx} \mid \text{rel} \\
\textit{(Expr)} & e & ::= & r \mid v \mid e + e \mid e - e \mid e * e \\
\textit{(Instr)} & c & ::= & r := \mathsf{x}_{o_r} \mid \mathsf{x}_{o_w} := e \mid r := \mathtt{CAS}_{o_r,o_w}(\mathsf{x}, e_r, e_w) \\
 & & & \mid \mathtt{skip} \mid r := e \mid \mathtt{print}(e) \\
\textit{(BBlock)} & B & ::= & c, B \mid \mathtt{jmp}\ \mathsf{f} \mid \mathtt{be}\ e, \mathsf{f}_1, \mathsf{f}_2 \mid \mathtt{call}(\mathsf{f}, \mathsf{f}_{ret}) \mid \mathtt{return} \\
\textit{(Cdhp)} & C & \in & \textit{Lab} \rightharpoonup \textit{BBlock} \\
\textit{(Code)} & \pi & ::= & \{\mathsf{f}_1 \leadsto C_1, \ldots, \mathsf{f}_k \leadsto C_k\} \\
\textit{(Prog)} & \mathbb{P} & ::= & \textbf{let}\ (\pi, \iota)\ \textbf{in}\ \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n
\end{array}
$$

Figure 7. Syntax of the CSimpRTL language.

also ensures the preservation of ww-RF (via ① and ③), allowing us to vertically compose verified optimizations.

Since we are concerned about verifying Opt, we need to prove the thread-local simulations for *all* source code. For this, we follow CompCert, which performs induction on the program structure of the source. We have applied our verification method Verif(Opt) to four optimization algorithms, including constant propagation (`ConstProp`), dead code elimination (`DCE`), common subexpression elimination (`CSE`) and loop invariant code motion (`LICM`). For `LICM`, we verify the two passes `LInv` and `CSE` separately, and conclude the correctness of `LICM` by transitivity of the refinement.

3 Preliminaries: The Promising Semantics

Figure 7 shows the syntax of our concurrent programming language CSimpRTL. A program $\mathbb{P}$ consists of n threads, each calls a function f in π. The code heap C of the function π(f) maps labels to basic blocks. Each basic block B is a sequence of instructions ending with a jump (either an unconditional jump `jmp`, a conditional jump `be`, an internal function call `call` or a `return` command).

An instruction c can be read, write, and compare-and-swap (`CAS`) operations on variables (memory locations), annotated with *access modes* o_r and/or o_w. We support three kinds of memory accesses on atomic locations: *relaxed* (rlx), *release writes* (rel) and *acquire reads* (acq). Reads and writes on non-atomic locations must be in the *non-atomic* (na) mode. The `CAS` instruction can only access an atomic location, and it carries two access modes, one for the read part and one for the write part. Besides, we also support fence instructions[1], and local computation ($r := e$) and `print` instructions that involve only registers. We require programmers to explicitly specify which variables are atomic, using the set ι, which plays a similar role as the keyword "`_Atomic`" for atomic variable declarations in C11 programs.

[1]To simplify the presentation, we omit fence operations (including release/acquire/sc fences) as well as the components in PS2.1 that model their semantics in the paper. We consider them and the full PS2.1 model in our supplementary appendix and Coq implementation [25].

$$
\begin{array}{rrcl}
\textit{(Time)} & f, t & \in & \mathbb{Q} \quad \textit{(Tid)}\ \mathsf{t} \in \mathbb{N} \quad \textit{(LocalState)}\ \sigma ::= \ldots \\
\textit{(TimeMap)} & T & \in & \textit{Var} \to \textit{Time} \quad \textit{(View)}\ V ::= (T_{\mathsf{na}}, T_{\mathsf{rlx}}) \\
\textit{(Message)} & m & ::= & \langle \mathsf{x} : v@(f,t], V\rangle \mid \langle \mathsf{x} : (f,t]\rangle \\
\textit{(Mem)} & M, P & \in & \mathcal{P}(\textit{Message}) \quad \textit{(ThrdState)}\ \textit{TS} ::= (\sigma, V, P) \\
\textit{(ThrdPool)} & \mathcal{TP} & \in & \textit{Tid} \to \textit{ThrdState} \\
\textit{(World)} & W & ::= & (\mathcal{TP}, \mathsf{t}, M) \\
\textit{(ThrdEvt)} & \textit{te} & ::= & \tau \mid \mathsf{out}(v) \mid \mathsf{R}(o_r, \mathsf{x}, v) \mid \mathsf{W}(o_w, \mathsf{x}, v) \\
 & & & \mid \mathsf{U}(o_r, o_w, \mathsf{x}, v_r, v_w) \mid \mathsf{prm} \mid \mathsf{ccl} \mid \mathsf{rsv} \\
\textit{(ProgEvt)} & \textit{pe} & ::= & \tau \mid \mathsf{out}(v) \mid \mathsf{sw} \\
\textit{(EvtTrace)} & \mathcal{B} & ::= & \epsilon \mid \textbf{done} \mid \textbf{abort} \mid \mathsf{out}(v) :: \mathcal{B}
\end{array}
$$

Figure 8. Machine states and events.

The program behaviors are defined following the promising semantics presented in [5], which we refer to as PS2.1.

Machine states. Figure 8 defines the machine states. The whole program configuration W consists of the shared memory M, a thread pool $\mathcal{TP}$ containing the local states of threads, and the thread id t of the current thread.

The global *shared memory* M keeps all the historical writes, which are *time-stamped messages* m in the form of $\langle \mathsf{x} : v@(f,t], V\rangle$. The message records the write of x with the value v. The *timestamp interval* $(f,t]$ denotes the range of timestamps from f (exclusive) to t (inclusive). Usually we use the "to"-timestamp t to identify the messages, but we also need the "from"-timestamp f to form the interval, which is needed to prevent two successful CAS from seeing the same write, as explained below. The message m also contains a *message view* V to model the synchronization between release writes and acquire reads, which is also explained in detail below. We denote the components of m by m.var, m.val, m.from, m.to and m.view.

To ensure the coherence of reads, each thread maintains a *view* V as part of its local state *TS*. It records, for each variable, the timestamp of the most recent write the thread has seen. So V contains two *time maps* T_{na} and T_{rlx} for non-atomic reads and relaxed reads, respectively. Each maps variables to the corresponding timestamps. A thread can only see the writes whose "to"-timestamp is no less than the timestamp in its local view. We use $T_1 \sqcup T_2$ (and lifted to $V_1 \sqcup V_2$) to "join" views (pointwise maximum). We also use $T^0 \triangleq \{\mathsf{x} \rightsquigarrow 0 \mid \mathsf{x} \in \textit{Var}\}$ and $V_\perp \triangleq (T^0, T^0)$ to represent the bottom time map and view respectively.

A thread can *promise a future write* by putting a write message into its local *promise set* P in *TS* and the memory M. The promise can be made non-deterministically at any step and it does not have to correspond directly to any specific write instruction, but it needs to be fulfilled by the thread in the future, by executing an actual write instruction. When fulfilled, the promise is removed from P.

Thread steps and memory operations. We write $\iota \vdash (\textit{TS}, M) \xrightarrow{\textit{te}} (\textit{TS}', M')$ for one thread step. The thread event *te* defined in Fig. 8 labels the operation. If *te* is not an out(v) event generated by a `print` instruction, we can omit *te* from the transition and view the step *silent*.

If $\textit{te} = \mathsf{W}(o_w, \mathsf{x}, v)$, the thread executes $\mathsf{x}_{o_w} := e$. It either *fulfills a promise* $m = \langle \mathsf{x} : v@(f,t], V\rangle$ and removes it from the thread's promise set *TS.P*, or generates a new message $m = \langle \mathsf{x} : v@(f,t], V\rangle$ and puts it into memory. The timestamp interval is non-deterministically chosen. It needs to be disjoint with the timestamp intervals of other messages. Also the "to"-timestamp t needs to be strictly larger than the one recorded for x on the thread's relaxed view, that is, $\textit{TS}.V.T_{\mathsf{rlx}}(\mathsf{x}) < t$. Then, the thread updates its view (both T_{na} and T_{rlx} in *TS.V*) on x, since t becomes its largest known timestamp. If the write is a *release write* ($o_w = \mathsf{rel}$), the message m generated will take the thread's current view as the message view (i.e. m.view = *TS.V*), so that when an *acquire read* of this write takes place, the reading thread can update its local view by merging it with this message view, establishing synchronization between the release write and the acquire read. The *non-atomic* and *relaxed* writes are non-synchronizing memory accesses and the messages generated by their executions take $V_\perp$ as the message view.

If $\textit{te} = \mathsf{R}(o_r, \mathsf{x}, v)$, the thread performs $r := \mathsf{x}_{o_r}$. The thread picks a concrete message $\langle \mathsf{x} : v@(f,t], V\rangle \in M$ to read, but t must be at least as large as the one recorded for x in the thread's view. Namely, $\textit{TS}.V.T_{\mathsf{na}}(\mathsf{x}) \le t$ if $o_r = \mathsf{na}$, or $\textit{TS}.V.T_{\mathsf{rlx}}(\mathsf{x}) \le t$ if $o_r \in \{\mathsf{rlx}, \mathsf{acq}\}$. Then the thread updates its view (both T_{na} and T_{rlx} in *TS.V* if $o_r \in \{\mathsf{rlx}, \mathsf{acq}\}$, or just T_{rlx} if $o_r = \mathsf{na}$) on x to record the new timestamp t. The resulting view V' is set to the new thread local view for the *non-atomic* or *relaxed* read. For the *acquire* read, the new thread local view becomes the join of V' and the message view V, i.e. $V' \sqcup V$.

The execution of the atomic update $r := \mathsf{CAS}_{o_r, o_w}(\mathsf{x}, e_r, e_w)$ generates the $\mathsf{U}(o_r, o_w, \mathsf{x}, v_r, v_w)$ event. It can be viewed as a combination of a read and a write. For the generated message $\langle \mathsf{x} : v_w@(f,t], V\rangle$, the "from"-timestamp f needs to be equal to the "to" timestamp of the message from which the CAS reads. Here we need the interval $(f,t]$ to ensure that two concurrent CAS cannot both succeed and read the same write. Consider the following example.

$$r_1 := \mathsf{CAS}(\mathsf{x}, 0, 1) \parallel r_2 := \mathsf{CAS}(\mathsf{x}, 0, 1)$$

Assume the memory only contains the initial message $m_0 = \langle \mathsf{x} : 0@(0,0], \_\rangle$. The semantics ensures that only one CAS can succeed. If the CAS of t_1 succeeds and generates a new message $\langle \mathsf{x} : 1@(0,t], \_\rangle$. Then the CAS of t_2 cannot succeed, otherwise it would also generate a message with a "from"-timestamp of 0, violating the requirement that the timestamp of all messages must be disjoint.

The promise step generates the prm event. As explained above, it adds a write message m into the thread's promise

$$\frac{\mathsf{t}' \in \mathrm{dom}(\mathcal{TP})}{\iota \vdash (\mathcal{TP}, \mathsf{t}, M) \overset{\mathsf{sw}}{\Longrightarrow} (\mathcal{TP}, \mathsf{t}', M)} \text{(sw-step)}$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), M) \longrightarrow^{+} (TS', M') \quad \mathsf{consistent}(TS', M', \iota)}{\iota \vdash (\mathcal{TP}, \mathsf{t}, M) \overset{\tau}{\Longrightarrow} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, M')} (\tau\text{-step})$$

$$\frac{\iota \vdash (\mathcal{TP}(\mathsf{t}), M) \xrightarrow{\mathsf{out}(v)} (TS', M')}{\iota \vdash (\mathcal{TP}, \mathsf{t}, M) \overset{\mathsf{out}(v)}{\Longrightarrow} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, M')} \text{(out-step)}$$

Figure 9. Interleaving machine steps.

set and the memory. Note that only non-atomic and relaxed writes can be promised.

The rsv and ccl events represent the creation and cancellation of a reservation. The reserve step (te = rsv) puts a special reservation message $\langle \mathsf{x} : (f, t] \rangle$ (see Fig. 8) into the thread's promise set and the memory. It allows the thread to reserve a timestamp interval that it plans to use later and prevents other threads from using it. When the thread wants to use its reserved timestamp interval, it takes a cancel step (te = ccl) to remove the reservation. The τ step represents a silent thread step with no memory effect (e.g. $r := e$).

Promise certification. All the promises made by a thread should be certified through the following *consistency check*:

$$\mathsf{consistent}(TS, M, \iota) \text{ iff } \exists TS'.\ \iota \vdash (TS, \widehat{M}) \longrightarrow^{*} (TS', \_) \ \wedge\ TS'.P = \emptyset$$

It requires that the current thread should be able to fulfill all its promises ($TS'.P = \emptyset$ for the final TS') if it executes in isolation, starting form the capped version $\widehat{M}$ of the current memory M. The *capped memory* $\widehat{M}$ is constructed from M in two steps: first we fill all the "gaps" between the timestamp intervals of the messages for the same location in M by inserting reservations in between, and then for every location x we insert a "cap reservation" $\langle \mathsf{x} : (t, t+1] \rangle$ in the memory, where t is the "to"-timestamp of the latest message on location x.

Machine steps. Execution of the whole program follows an interleaving semantics. As the (τ-step) rule in Fig. 9 shows, a machine step may consist of multiple silent thread steps (i.e. steps without producing the out(v) event, which are steps that do not execute the `print` instruction), until it reaches a *consistent* configuration (defined above). The overall machine step is labeled with $pe = \tau$ (see Fig. 8 for the definition of pe), saying that it is not externally observable.

A context switch can be done non-deterministically at any machine step by resetting the current thread id (the (sw-step)). An (out-step) executes `print` and produces an externally observable out(v) event.

Behaviors. The behaviors of a program are modeled as a set of observable event traces. As defined in Fig. 8, an observable event trace $\mathcal{B}$ is a finite sequence of the output event (out(v)) and may end with a termination marker **done** or an abortion marker **abort**. We use $\mathbb{P} \subseteq \mathbb{P}'$ to represent the event-trace refinement, which means that the set of observable event traces generated by the execution of $\mathbb{P}$ is the subset of one generated by the execution of $\mathbb{P}'$. $\mathbb{P} \approx \mathbb{P}'$ represents that the executions of $\mathbb{P}$ and $\mathbb{P}'$ generate the same set of observable event traces.

$$
\begin{array}{llcl}
(\mathit{NPProg}) & \hat{\mathbb{P}} & ::= & \mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n \\
(\mathit{SwBit}) & \beta & ::= & \circ \mid \bullet \qquad (\mathit{NPWorld})\ \hat{W} ::= (\mathcal{TP}, \mathsf{t}, M, \beta) \\
(\mathit{NA}) & na & ::= & \tau \mid \mathsf{R}(\mathsf{na}, \mathsf{x}, v) \mid \mathsf{W}(\mathsf{na}, \mathsf{x}, v) \\
(\mathit{PRC}) & prc & ::= & \mathsf{prm} \mid \mathsf{rsv} \mid \mathsf{ccl} \\
(\mathit{AT}) & at & \in & \{te \mid te \notin (\mathit{NA} \cup \mathit{PRC})\}
\end{array}
$$

$$\frac{\begin{array}{c}\iota \vdash (TS, M) \xrightarrow{te} (TS', M') \\ te \in NA \implies \beta' = \bullet \qquad te \in AT \implies \beta' = \circ \\ te \in \{\mathsf{prm}, \mathsf{rsv}\} \implies \beta = \beta' = \circ \qquad te = \mathsf{ccl} \implies \beta = \beta'\end{array}}{\iota \vdash (TS, M, \beta) \overset{te}{\longmapsto} (TS', M', \beta')}$$

$$\frac{\begin{array}{c}\iota \vdash (\mathcal{TP}(\mathsf{t}), M, \beta) \longmapsto^{+} (TS', M', \beta') \\ \mathsf{consistent}_{\mathsf{NP}}(TS', M', \beta', \iota)\end{array}}{\iota \vdash (\mathcal{TP}, \mathsf{t}, M, \beta) \overset{\tau}{:\Longrightarrow} (\mathcal{TP}\{\mathsf{t} \rightsquigarrow TS'\}, \mathsf{t}, M', \beta')}$$

$$\frac{\mathsf{t}' \in \mathrm{dom}(\mathcal{TP})}{\iota \vdash (\mathcal{TP}, \mathsf{t}, M, \circ) \overset{\mathsf{sw}}{:\Longrightarrow} (\mathcal{TP}, \mathsf{t}', M, \circ)}$$

Figure 10. Core rules of our non-preemptive semantics.

4 Non-Preemptive Semantics

Figure 10 shows our non-preemptive version of the promising semantics. Its main feature is that we disallow context switches and promise/reserve steps after non-atomic reads and writes. Therefore, execution of a block B of statements consisting of only non-atomic accesses cannot be interrupted by other threads. However, for the following reasons this requirement does *not* make B sequential or "atomic" in the traditional sense. First, the non-atomic steps in B still follow the promising semantics, therefore each non-atomic read may see multiple writes in M, according to the thread's local view. Second, before entering the block B, we can still make promises corresponding to the non-atomic writes in B. Context switches between these promise steps and the execution of B are permitted. These are the keys to make the non-preemptive semantics equivalent to the interelaving PS.

In detail, we write **let** (π, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$ or $\hat{\mathbb{P}}$ for the program with the non-preemptive semantics, to distinguish it from the one with the interleaving semantics. The machine state $\hat{W}$ is defined similarly as W in Fig. 8, but is extended with a "switch bit" β to indicate whether a switch step is allowed ($\circ$) or not ($\bullet$). We classify the operations into three groups, NA, PRC and AT. NA operations refer to non-atomic memory accesses, and operations that have no memory or synchronization effects. PRC refers to promise (prm), reserve

$$\frac{\begin{array}{c}\mathcal{TP}(\mathsf{t}) = (\sigma, V, P) \quad \mathsf{nxt}(\sigma) = \mathsf{W}(\mathsf{na}, \mathsf{x}, \_) \\ m \in (M \backslash P) \quad m.\mathsf{var} = \mathsf{x} \quad V.T_{\mathsf{rlx}}(\mathsf{x}) < m.\mathsf{to}\end{array}}{(\mathcal{TP}, \mathsf{t}, M) \Longmapsto \mathsf{ww\text{-}Race}}$$

$$\frac{\begin{array}{c}\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n \overset{init}{\Longrightarrow} W \\ \iota \vdash W \Longrightarrow^* W' \quad W' \Longmapsto \mathsf{ww\text{-}Race}\end{array}}{\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n \Longmapsto \mathsf{ww\text{-}Race}} \qquad \frac{\neg(\mathbb{P} \Longmapsto \mathsf{ww\text{-}Race})}{\mathsf{ww\text{-}RF}(\mathbb{P})}$$

Figure 11. Write-write race freedom in PS.

(rsv) and cancel (ccl) steps. *AT* refers to other atomic memory accesses and synchronization steps.

The first rule in Fig. 10 shows the allowed combination of the switch bit and the thread step. It reuses the thread step $\iota \vdash (TS, M) \xrightarrow{te} (TS', M')$ in the promising semantics. The switch bit turns off if *te* is a *NA* step, and turns on if it is an *AT* step. The promise and reserve steps are allowed only when the switch bit is turned on, and they do not change the switch bit (i.e. $\beta = \beta' = \circ$). The cancel steps are allowed at any places and do not change the switch bit either.

As the second rule of Fig. 10 shows, we lift the new silent thread steps to a machine step, by requiring that the thread should reach a consistent configuration, similar to the (τ-step) of the interleaving semantics in Fig. 9. Here consistent$_{\mathsf{NP}}$ is defined the same as consistent but using the new thread step instead. Other kinds of thread steps are also lifted to machine steps in the same way as in Fig. 9 and we omit the details. Different from the interleaving semantics, here we only switch to another thread when $\beta = \circ$, as shown by the last rule of Fig. 10.

We prove Thm. 4.1 (⑤ in Fig. 6), saying that a program in our non-preemptive semantics generates the same observable behaviors as in the original interleaving PS2.1.

Theorem 4.1 (Semantics Equivalence)**.** $\forall \pi, \mathsf{f}_1, \ldots, \mathsf{f}_n, \iota.$

$$\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n \approx \mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n.$$

One may question this equivalence and argue that, without interleavings inside a block of non-atomic accesses, the non-preemptive semantics would disallow the following behaviors easily produced by an interleaving semantics:

(1) redundant reads can see different values; and
(2) redundant writes can all be seen by other threads.

Our non-preemptive semantics *can* produce these behaviors because, for (1), by reusing the thread steps of PS2.1, a read needs not read the "latest" write; and for (2), all the writes can be promised before entering the block of non-atomic accesses, so that other threads can see them.

5 Write-Write Race Freedom

As explained in Sec. 2.4, we assume the source programs are free of write-write races. We define write-write race freedom (ww-RF) in PS2.1 in Fig. 11. For a program $\mathbb{P}$, ww-RF($\mathbb{P}$) holds, if its execution never reaches a machine state W' that generates a write-write race. The machine state $W = (\mathcal{TP}, \mathsf{t}, M)$ generates a write-write race ($W \Longmapsto$ ww-Race), if the current thread t can take a non-atomic write step on some location x (we use nxt(σ) to get the next operation of the program in the following execution) while there exists a message $m \in (M \backslash (\mathcal{TP}(\mathsf{t}).P))$ on x that has not been observed by t. In other words, there is a write on x (represented by m) concurrently with the non-atomic write by t.

We define ww-NPRF($\hat{\mathbb{P}}$) similarly, which says that the execution of $\hat{\mathbb{P}}$ in the *non-preemptive* semantics will not reach a machine state generating write-write races. We prove that ww-NPRF is equivalent to ww-RF in Lm. 5.1 (① in Fig. 6).

Lemma 5.1 (ww-RF Equivalence)**.** $\forall \pi, \mathsf{f}_1, \ldots, \mathsf{f}_n, \iota.$

$$\begin{array}{l}\mathsf{ww\text{-}RF}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n) \\ \qquad \Longleftrightarrow \mathsf{ww\text{-}NPRF}(\mathbf{let}\ (\pi, \iota)\ \mathbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n).\end{array}$$

To vertically compose verified optimizations, the target program of an optimization needs to satisfy ww-RF as well, since it can be the input source of a subsequent optimization. As such, our verification method (i.e. the simulation in Sec. 6) should ensure the preservation of ww-RF.

6 Thread-Local Simulation

In this section, we define a *thread-local* simulation as the formal correctness definition of optimizations. It can be viewed as a specialization of the standard simulation (see Fig. 2(a)) in the setting of PS2.1, but it achieves both of the two challenging goals: 1) it is parallel compositional; 2) it preserves write-write race freedom. For 1), our simulation carries the invariant parameter I relating the source and target shared states at switch points (Sec. 6.1). The switch points are determined by the non-preemptive semantics in Sec. 4. For 2), we introduce the delayed write set $\mathcal{D}$ updated along with the execution steps inside the simulation (Sec. 6.2). We end this section with the optimization correctness theorem (Sec. 6.3).

6.1 Invariant Parameter

As explained in Sec. 2.2, the invariant parameter I specifies the shared states at switch points. It is the key to ensure horizontal compositionality. We allow it to be instantiated differently when verifying different optimizations. The type of I is shown in Fig. 12. Users instantiating $I(\varphi, \mathbb{S}, \iota)$ are expected to specify the application-dependent invariant over φ and $\mathbb{S}$, with the help of the set ι of atomic variables for the code being verified. Here φ is the timestamp mapping explained soon, and the shared state $\mathbb{S} = (M_t, M_s)$ consists of the memories (M) at the target and source levels.

Timestamp mapping. To relate the messages in the target and source memories, we introduce a partial mapping φ called "timestamp mapping" whose type is defined at the top of Fig. 12. $\varphi(\mathsf{x}, t) = t'$ maps the "to"-timestamp t at the target level to the "to"-timestamp t' at the source for location x.

$$
\begin{array}{rrcl}
(TMap) & \varphi & \in & (Var \times Time) \rightharpoonup Time \\
(Sst) & \mathbb{S} & \triangleq & (M_t, M_s) \quad where\ M_t, M_s \in Mem \\
(Inv) & I & \in & TMap \to Sst \to Atms \to Prop
\end{array}
$$

$$
\begin{array}{rcl}
\lfloor\!\lfloor M \rfloor\!\rfloor & \triangleq & \{(\mathsf{x}, t) \mid \langle \mathsf{x} : \_@(\_, t], \_\rangle \in M\} \\
\varphi(M) & \triangleq & \{(\mathsf{x}, t') \mid \exists t.\ (\mathsf{x}, t) \in \lfloor\!\lfloor M \rfloor\!\rfloor \wedge \varphi(\mathsf{x}, t) = t'\} \\
\mathsf{mon}(\varphi) & \triangleq & \forall \mathsf{x}, t_1, t_2.\ (t_1 < t_2) \wedge (\{(\mathsf{x}, t_1), (\mathsf{x}, t_2)\} \subseteq \mathrm{dom}(\varphi)) \\
 & & \quad \Longrightarrow \varphi(\mathsf{x}, t_1) < \varphi(\mathsf{x}, t_2) \\
\mathsf{wf}(I, \iota) & \triangleq & I(\varphi_0, (M_0, M_0), \iota) \\
 & & \wedge (\forall \varphi, M_t, M_s.\ I(\varphi, (M_t, M_s), \iota) \Longrightarrow \\
 & & \quad \mathrm{dom}(\varphi) = \lfloor\!\lfloor M_t \rfloor\!\rfloor \wedge \varphi(M_t) \subseteq \lfloor\!\lfloor M_s \rfloor\!\rfloor \wedge \mathsf{mon}(\varphi))
\end{array}
$$

Figure 12. Timestamp mapping φ and invariant I.

Initially when the thread starts the execution, the timestamp mapping is $\varphi_0 \triangleq \{(\mathsf{x}, 0) \rightsquigarrow 0 \mid \mathsf{x} \in Var\}$, for the initial memories $M_0 \triangleq \{\langle \mathsf{x} : 0@(0, 0], V_\perp \rangle \mid \mathsf{x} \in Var\}$. As the thread executes, the timestamp mapping φ may be expanded to relate timestamps for the new writes, but the φ-related memories M_t and M_s must always satisfy the user-specified invariant I at switch points.

To see how the invariant I imposes constraints on φ, M_t and M_s, we show a simple example:

$$
\begin{array}{l}
I_{id}(\varphi, (M_t, M_s), \iota) \triangleq (M_t = M_s) \\
\quad \wedge (\mathrm{dom}(\varphi) = \lfloor\!\lfloor M_t \rfloor\!\rfloor) \wedge (\forall (\mathsf{x}, t) \in \mathrm{dom}(\varphi).\ \varphi(\mathsf{x}, t) = t)
\end{array}
$$

I_{id} requires that the source and target memories be the same, and consequently φ be an identity timestamp mapping. Here $\lfloor\!\lfloor M \rfloor\!\rfloor$ collects the (x, t) pairs of concrete messages in M (defined in Fig. 12). Despite its simple look, I_{id} is applicable for many optimizations, including ConstProp and CSE.

***Well-formed** I.* Since I is provided by verifiers, it must pass some sanity check (under the atomic variable set ι for the code being verified), defined as $\mathsf{wf}(I, \iota)$ in Fig. 12. It has two requirements. First, I should hold over the initial φ_0 and M_0. Second, whenever $I(\varphi, (M_t, M_s), \iota)$ holds, each concrete message in M_t has a related concrete message in M_s through φ, and φ is monotonically increasing on timestamps ($\mathsf{mon}(\varphi)$). It is easy to check that $\mathsf{wf}(I_{id}, \iota)$ holds for the above I_{id} and any ι.

***Simulation with** I.* Our thread-local simulation between the target and source code π_t and π_s is written as $I, \iota \models \pi_t \preccurlyeq \pi_s$, defined in Def. 6.1.

Definition 6.1 (Thread-local upward simulation)**.**
$I, \iota \models \pi_t \preccurlyeq \pi_s$ iff (1) $\mathsf{wf}(I, \iota)$; and (2) for any σ_t and f, if $\mathsf{Init}(\pi_t, \mathsf{f}) = \sigma_t$, then there exists σ_s such that $\mathsf{Init}(\pi_s, \mathsf{f}) = \sigma_s$ and $I, \iota \models ((\sigma_t, V_\perp, \emptyset), M_0) \preccurlyeq^{\circ, \emptyset}_{\varphi_0} ((\sigma_s, V_\perp, \emptyset), M_0)$.

In addition to the sanity check $\mathsf{wf}(I, \iota)$, Def. 6.1 says, if the execution of a target thread starts from the function f in π_t in the initial state, the source thread can also start from f in π_s, and the initial thread configurations $((\sigma_t, V_\perp, \emptyset), M_0)$ and $((\sigma_s, V_\perp, \emptyset), M_0)$ have the simulation.

$$
\begin{array}{rrcl}
(Index) & i & \in & \ldots \qquad (DlyItem) \quad d \in (Var \times Time) \\
(Dlyset) & \mathcal{D} & \in & DlyItem \rightharpoonup Index
\end{array}
$$

$$
\mathcal{D}' < \mathcal{D} \triangleq \mathrm{dom}(\mathcal{D}') = \mathrm{dom}(\mathcal{D}) \wedge \forall d \in \mathrm{dom}(\mathcal{D}).\ \mathcal{D}'(d) < \mathcal{D}(d)
$$

$$
\frac{\begin{array}{l}
te = \mathsf{W}(\mathsf{na}, \mathsf{x}, v) \Longrightarrow \\
\quad \exists t.\ (\mathsf{x}, t) \in \lfloor\!\lfloor (M' - M) \cup (TS.P - TS'.P) \rfloor\!\rfloor \\
\quad \wedge \exists i.\ \mathcal{D}' = \mathcal{D} \uplus \{(\mathsf{x}, t) \rightsquigarrow i\} \\
te \neq \mathsf{W}(\mathsf{na}, \_, \_) \Longrightarrow \mathcal{D}' = \mathcal{D}
\end{array}}{((TS, M), te, (TS', M')) \vdash \mathcal{D} \rightsquigarrow \mathcal{D}'} \text{(tgt-D)}
$$

$$
\frac{\begin{array}{c}
\iota \vdash (TS, M) \xrightarrow{te} (TS', M') \\
te = \mathsf{W}(\mathsf{na}, \mathsf{x}, v) \Longrightarrow \mathcal{D}' = (\mathcal{D} \backslash (\mathsf{x}, t)) \\
te \neq \mathsf{W}(\mathsf{na}, \_, \_) \Longrightarrow \mathcal{D}' = \mathcal{D} \quad \text{side condition} \ldots
\end{array}}{\iota \vdash (TS, M, \mathcal{D}) \overset{te}{\twoheadrightarrow} (TS', M', \mathcal{D}')} \text{(src-D)}
$$

Figure 13. Delayed write set.

The simulation $I, \iota \models (TS_t, M_t) \preccurlyeq^{\beta, \mathcal{D}}_{\varphi} (TS_s, M_s)$ relates the target and source's thread configurations, (TS_t, M_t) and (TS_s, M_s). The parameter φ records the timestamp mapping at the last switch point, and the switch bit β indicates whether the thread can switch at the current point (yes, if $\beta = \circ$; and no, if $\beta = \bullet$). The delayed write set $\mathcal{D}$ will be explained in Sec. 6.2 and can be ignored for the moment.

As depicted in Fig. 2(b), I should hold at switch points (i.e. when $\beta = \circ$). The simulation says, whenever $I, \iota \models (TS_t, M_t) \preccurlyeq^{\circ, \mathcal{D}}_{\varphi} (TS_s, M_s)$, we have $I(\varphi, (M_t, M_s), \iota)$ and

$$
\begin{array}{l}
\forall \varphi', M_t', M_s'.\ I(\varphi', (M_t', M_s'), \iota) \wedge \mathsf{Rely}((M_t, M_s), (M_t', M_s'), \ldots) \\
\Longrightarrow I, \iota \models (TS_t, M_t') \preccurlyeq^{\circ, \mathcal{D}}_{\varphi'} (TS_s, M_s').
\end{array}
$$

That is, when switching back from the environment transition, the simulation relation is preserved as long as I still holds over the new shared state (M_t', M_s') and timestamp mapping φ', and the transition satisfies a Rely condition. Unlike the parameter I, the Rely condition encodes the semantics specific transitions, e.g. the write messages in M_t must also be in M_t' since messages can never be removed from the memory. Its definition is fixed as part of the simulation. We do not expand it here for brevity.

6.2 Delayed Write Set

To enforce the preservation of write-write race freedom (i.e. if the source program is race-free, so is the target; or say, if the target program is racy, so is the source), our thread-local simulation requires that all locations written by the target thread should also be written by the source. That said, we should also allow the source state to be temporarily "left behind" the target. That is, when the target thread does a write step or fulfills a promise, the source may not perform the corresponding step at present, but it must eventually do the step. For this, we introduce the delayed write set $\mathcal{D}$ to record the set of writes which must be caught up by the source thread later. As defined in Fig. 13, $\mathcal{D}$ maps each

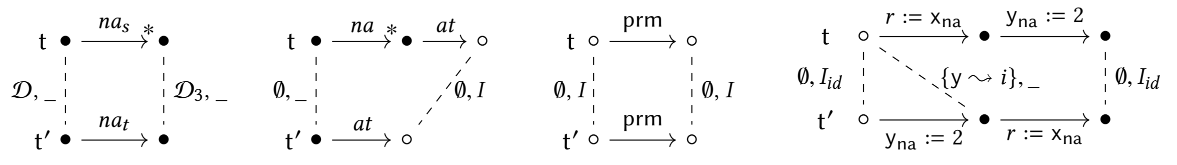

Figure 14. Simulation diagrams. (t′ and t represent the target and source threads respectively.)

delayed item d to a well-founded index i, where d is a pair (x, t) representing a non-atomic write.

More specifically, $\mathcal{D}$ plays two roles. First, $\mathcal{D}$ records all the non-atomic writes of the target. As defined in the (tgt-D) rule in Fig. 13, we write $((TS, M), te, (TS', M')) \vdash \mathcal{D} \rightsquigarrow \mathcal{D}'$ to compute the new $\mathcal{D}'$ from the original $\mathcal{D}$ for the target step $\iota \vdash (TS, M) \xrightarrow{te} (TS', M')$. When this target step is a non-atomic write (i.e. $te = \mathsf{W}(\mathsf{na}, \mathsf{x}, v)$), we first find out the corresponding write message (identified by the pair (x, t)), which must be either a newly added message in the memory $(M' - M)$ or a fulfilled promise in $(TS.P - TS'.P)$. Then (x, t) is put into $\mathcal{D}$ with some index i to get the new $\mathcal{D}'$.

Second, the well-founded indexes in $\mathcal{D}$ are used to ensure that the source eventually catches up the delayed writes *within finite steps*. An index assigned to a write (x, t) should be decreased for source steps which do not write to x. To formulate this idea, we first extend a *source* thread step with the writes remained to catch up, written as $\iota \vdash (TS, M, \mathcal{D}) \overset{te}{\twoheadrightarrow} (TS', M', \mathcal{D}')$. It is defined in the (src-D) rule in Fig. 13. We compute $\mathcal{D}'$ by removing from $\mathcal{D}$ the write performed by the source step (see the case for $te = \mathsf{W}(\mathsf{na}, \mathsf{x}, v)$). There is also a side condition (elided to simplify the presentation) ensuring that the source step does not read an originally unobserved write in $\mathcal{D}$, so it does not make an originally possible write-write race impossible. Then, in the simulation, we decrease the indexes of the remaining delayed writes (written as $\mathcal{D}' < \mathcal{D}$, defined at the top of Fig. 13).

In summary, the simulation $I, \iota \models (TS_t, M_t) \preccurlyeq_{\varphi}^{\beta, \mathcal{D}} (TS_s, M_s)$ requires that, if the target thread executes a non-atomic step, i.e. $(\iota \vdash (TS_t, M_t) \xrightarrow{te} (TS_t', M_t')) \wedge te \in NA$, then there exist TS_s', M_s', $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ such that:

- $((TS_t, M_t), te, (TS_t', M_t')) \vdash \mathcal{D} \rightsquigarrow \mathcal{D}_1$;
- $\iota \vdash (TS_s, M_s, \mathcal{D}_1) \overset{na}{\twoheadrightarrow}{}^{*} (TS_s', M_s', \mathcal{D}_2)$, $\mathcal{D}_3 < \mathcal{D}_2$;
- $I, \iota \models (TS_t', M_t') \preccurlyeq_{\varphi}^{\bullet, \mathcal{D}_3} (TS_s', M_s')$.

Simulation diagrams. We draw the simulation diagrams for the current thread's steps in Fig. 14(a-c). We have already discussed the case for non-atomic steps (Fig. 14(a)).

For atomic memory accesses (Fig. 14(b)), which may establish synchronizations between different threads, our simulation requires that the target and source threads always perform the same atomic memory access. Before that the source is allowed to do some non-atomic steps. This enables us to verify transformations like $(r := 1; \mathsf{x_{rlx}} := r) \rightsquigarrow \mathsf{x_{rlx}} := 1$. The delay write set should have been empty when taking the atomic step. After the step, the switch bit is ∘ and the invariant I needs to be reestablished.

If the target thread takes a promise step for a future write (Fig. 14(c)), the simulation requires the source thread to also make a promise for the corresponding future write. As required by our non-preemptive semantics, the switch bits must be ∘ before and after the promise steps, so the invariant I needs to be preserved. The cases for reserve and cancel steps are similar.

Example: instruction reordering. We prove the example of Reorder (in Sec. 2.3) satisfies our thread-local simulation. We instantiate the invariant parameter as I_{id} defined in Sec. 6.1, and build the simulation in Fig. 14(d). Here we don't let the source thread t execute when the target t′ executes $\mathsf{y_{na}} := 2$. At this point I_{id} is temporarily broken, but this is fine since I_{id} is required to hold at switch points only. We add y with some index i into the delayed write set to make sure that the source t will write to y in finite steps. Then, after t′ executes $r := \mathsf{x_{na}}$, we can let t read the corresponding message at the source, and execute the write to empty the delayed write set and reestablish I_{id}.

6.3 The Optimization Correctness Theorem

Via the proof path in Fig. 6, we can prove that our simulation ensures the optimization correctness.

We first present Lm. 6.2 (③ in Fig. 6): under the write-write race freedom assumption, our thread-local simulation defined in Def. 6.1 is compositional and preserves write-write race freedom. The whole program simulation $\hat{\mathbb{P}}_t \leqslant \hat{\mathbb{P}}_s$ relates the non-preemptive executions of the whole programs $\hat{\mathbb{P}}_t$ and $\hat{\mathbb{P}}_s$. Safe means that the execution of the whole program will not abort.

Lemma 6.2 (Compositionality and ww-RF Preserving)**.** For any π_t, π_s, ι, $\mathsf{f}_1, \ldots, \mathsf{f}_n$ and I, if

1. $I, \iota \models \pi_t \preccurlyeq \pi_s$,
2. Safe(**let** (π_s, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$),
3. ww-NPRF(**let** (π_s, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$),

then **let** (π_t, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n \leqslant$ **let** (π_s, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$ and ww-NPRF(**let** (π_t, ι) **in** $\mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n$).

An optimizer Opt takes the source code π_s and the set ι of atomic variables as input, and returns the target code π_t.

We do not perform optimizations on atomic variables, so the set of atomic variables in the target must still be ι. Opt is verified, denoted by Verif(Opt), if we can always establish the thread-local simulation between any pairs of π_t and π_s.

Definition 6.3 (Verified optimizer). Verif(Opt) iff

$$\forall \pi_t, \pi_s, \iota.\ \mathsf{Opt}(\pi_s, \iota) = \pi_t \implies \exists I.\ \ I, \iota \models \pi_t \preccurlyeq \pi_s.$$

An Opt is correct (Def. 6.4), if the refinement holds for all possible source and target programs. We prove Thm. 6.5 that a verified optimizer is correct, via the proof path in Fig. 6. Detailed proofs are given in the supplementary appendix [25].

Definition 6.4 (Correctness of optimizer).
Correct(Opt) iff, for any π_s, π_t, $\mathsf{f}_1, \ldots, \mathsf{f}_n$ and ι, if

1. $\mathsf{Opt}(\pi_s, \iota) = \pi_t$,
2. ww-RF(**let** (π_s, ι) **in** $\mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n$),
3. Safe(**let** (π_s, ι) **in** $\mathsf{f}_1 \parallel \cdots \parallel \mathsf{f}_n$),

then **let** (π_t, ι) **in** $\mathsf{f}_1 \| \ldots \| \mathsf{f}_n \subseteq$ **let** (π_s, ι) **in** $\mathsf{f}_1 \| \ldots \| \mathsf{f}_n$.

Theorem 6.5 (Optimization correctness theorem).

$$\forall \mathsf{Opt}.\ \mathsf{Verif(Opt)} \implies \mathsf{Correct(Opt)}.$$

By Thm. 6.5, we prove that constant propagation, common subexpression elimination, dead code elimination and loop invariant code motion are all correct in PS2.1.

Theorem 6.6 (Correct Optimizers).
Correct(ConstProp)∧Correct(CSE)∧Correct(DCE)∧Correct(LICM).

7 Verified Optimizations

As an example, in Sec. 7.1 we briefly explain the verification of dead code elimination. The detailed proofs and the verification of the other optimization algorithms are given in the supplementary appendix [25]. In Sec. 7.2, we analyze which optimizations are supported and which are not.

7.1 Dead Code Elimination

Dead code elimination (DCE) eliminates writes to dead variables. A variable is called *dead* (or say, *not live*) at a program point if its value is not used later in any (sequential) execution of the code. Following CompCert, we define DCE as:

$$\mathsf{DCE}(\pi_s, \iota) \ \triangleq\ \mathsf{Translater}_{dce}(\pi_s, A_l) \quad \textit{where}\ \ A_l = \mathsf{Lv\_Analyzer}(\pi_s)$$

It relies on the results of liveness analysis Lv_Analyzer, which computes the set $L_{nl} \subseteq (\mathit{Var} \cup \mathit{Reg})$ of dead variables and registers at every program point. It is a backward analysis: for each instruction c, it computes L'_{nl} before c, when given L_{nl} after c. Then, $A_l = \mathsf{Lv\_Analyzer}(\pi_s)$ collects the results L_{nl} for all the program points of π_s. The code transformation $\mathsf{Translater}_{dce}$ applies the single-instruction transformation Transl_d on all instructions of π_s. Given the analysis result L_{nl} *after* c, $\mathsf{Transl}_d(c, L_{nl})$ transforms c to skip (i.e. eliminates c) if c is a write to a non-atomic location x (or a register r)

```
y_na := 2;                            g() { int r1,r2;
 {y}                                    r1 := x_acq;
x_rel := 1;   Unsafe   skip;            if(r1 == 1) {
 {y}            ~>     x_rel := 1;        r2 := y_na;
y_na := 4;             y_na := 4;         print(r2); }
 {}                                   }
```

Figure 15. DCE is unsafe across release writes.

which is dead after c, i.e. $\mathsf{x} \in L_{nl}$ (or $r \in L_{nl}$); and keeps c unchanged otherwise.

In PS, it is unsound to perform DCE across release writes. We forbid it by defining Lv_Analyzer to say that no variable is dead before a release write. To see how this works, consider the example in Fig. 15. We annotate the left code (the source) with the set L_{nl} of dead variables for each program point in blue (or in red, for the incorrect L_{nl}). Starting from the bottom, we assume L_{nl} is empty at the end of this code. Then, y is dead before $\mathsf{y}_{\mathsf{na}} := 4$. An incorrect liveness analysis would keep viewing y as dead before the release write (see the red annotation). Then the transformation would eliminate the first write to y. The problem is, g() may output the initial value 0 of y when running in parallel with the target code, but can only output 2 or 4 with the source due to the release-acquire synchronization.

Intuitively, the release write can synchronize with the other thread's acquire read, so that the variables' values before the release write are guaranteed to be seen by the other thread. Thus these variables cannot be viewed dead.

By contrast, it is sound to perform DCE across relaxed writes and atomic (acquire/relaxed) reads as well as non-atomic reads and writes.

Lemma 7.1 (DCE is verified). Verif(DCE).

We prove the lemma by showing that

$$\forall \pi_t, \pi_s, \iota.\ \mathsf{DCE}(\pi_s, \iota) = \pi_t \implies I_{dce}, \iota \models \pi_t \preccurlyeq \pi_s.$$

In the proof, we verify Lv_Analyzer and $\mathsf{Translater}_{dce}$ separately. Lv_Analyzer is verified following the abstract interpretation framework in CompCert. For $\mathsf{Translater}_{dce}$, we do induction on the program structure of the source. We establish the simulation relations for single instructions (the base cases of the induction), and derive the simulations for *all* source code by applying the induction hypothesis and using the compositionality of the simulation.

To establish the simulation, we instantiate the invariant parameter I of the simulation as I_{dce} defined below.

$$\begin{array}{l} I_{dce}(\varphi, (M_t, M_s), \iota) \ \triangleq\ (\varphi, \iota \vdash M_t \sim M_s) \wedge \\ \quad (\forall \mathsf{x} \notin \iota, t > 0.\ \langle \mathsf{x} : \_@(\_, t], \_\rangle \in M_t \\ \qquad\qquad \implies \exists \langle \mathsf{x} : \_@(f', t'], \_\rangle \in M_s, t_r. \\ \qquad\qquad\quad \varphi(\mathsf{x}, t) = t' \wedge t_r < f' \wedge \\ \qquad\qquad\quad (\forall m \in M_s(\mathsf{x}).\ m.\mathsf{to} \le t_r \vee t' \le m.\mathsf{from})) \end{array}$$

Besides the side condition $(\varphi, \iota \vdash M_t \sim M_s)$ (definition omitted for brevity) ensuring that I_{dce} is well-formed $(\mathsf{wf}(I_{dce}, \iota))$,

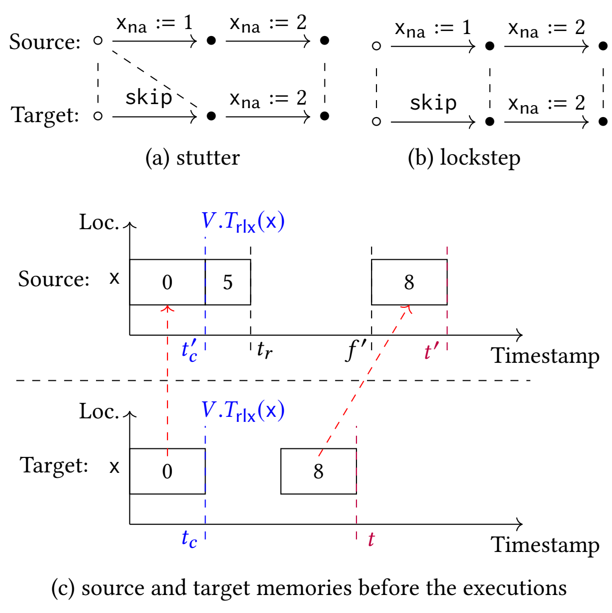

Figure 16. Simulations for a small example of DCE.

I_{dce} mainly requires that, for every message m_t representing a concrete write to a non-atomic location x in M_t, it has a φ-related message m_s in M_s, and there exists an unused timestamp interval $(t_r, f']$ just before m_s (where f' is m_s's "from"-timestamp). Here $M_s(\mathtt{x})$ denotes a memory consisting of only the messages with location x in M_s.

We need the unused timestamp interval to allow the source thread to perform dead writes. To see why, consider the following source and target code for a single thread:

$$\mathtt{x_{na}} := 1; \mathtt{x_{na}} := 2; \quad \overset{DCE}{\rightsquigarrow} \quad \mathtt{skip}; \mathtt{x_{na}} := 2; \tag{1}$$

To build the simulation between them, we have two options, shown in Fig. 16(a) and (b) respectively. The first one is to use a stutter simulation in Fig. 16(a), where we let the source stutter when the target does skip, and let the source execute till the end for the target's single write. The stutter simulation works well for this simple example, but will encounter problems for transformations in the following form:

$$\mathtt{x_{na}} := 1; c_1; \dots; c_n; \mathtt{x_{na}} := 2; \quad \overset{DCE}{\rightsquigarrow} \quad \mathtt{skip}; c_1; \dots; c_n; \mathtt{x_{na}} := 2;$$

When there are many other instructions $c_1; \dots; c_n$ between the dead write $\mathtt{x_{na}} := 1$ and the next write to x, should we let the source still stutter when the target executes $c_1; \dots; c_n$? If yes, we would have to remember all the steps that the source is left behind, which seems complex and impractical.

In our proof of DCE, we choose the alternative, the lockstep simulation in Fig. 16(b), where the source executes the dead write when the target does skip. The question is, which timestamp should we assign to the dead write of the source?

As a concrete example, suppose the memories shown in Fig. 16(c) are before the source and target threads execute the code in (1). For x, the source memory contains three messages while the target contains two (the extra message at the source represents a dead write eliminated at the target). We draw a box for each message, where the value inside the box is the message's written value to x, and draw a red dashed arrow to relate a target message to the corresponding source one. The current views on x (i.e. $V.T_{\mathsf{rlx}}(\mathtt{x})$) of the source and target threads are at the blue timestamps t_c' and t_c respectively. Now, following the lockstep simulation in Fig. 16(b), the source thread executes $\mathtt{x_{na}} := 1$ generating a message $\boxed{1}$ while the target executes skip. Shall we insert $\boxed{1}$ to the left or right side of the message $\boxed{8}$?

Our answer is, $\boxed{1}$ cannot be inserted to the right of $\boxed{8}$. To see why, consider the next steps executing $\mathtt{x_{na}} := 2$ by the source and target threads. Since the target thread view is at t_c, it is possible that the new message $\boxed{2}$ generated by the target thread has a lower timestamp than $\boxed{8}$, thus is inserted in between $\boxed{0}$ and $\boxed{8}$. But for the source, since it executes $\mathtt{x_{na}} := 2$ after $\mathtt{x_{na}} := 1$, its message $\boxed{2}$ should have a greater timestamp than $\boxed{1}$ and be inserted to the right of $\boxed{1}$. If $\boxed{1}$ at the source is inserted to the right of $\boxed{8}$, we will see that $\boxed{2}$ and $\boxed{8}$ are in different orders at the source and target! This is problematic, because if the thread reads x after the code of (1), it must obtain 2 at the source since $\boxed{2}$ is the rightmost message, but can obtain either 2 or 8 at the target.

Thus, we insert $\boxed{1}$ in between $\boxed{5}$ and $\boxed{8}$ at the source. To ensure there indeed exists "space" to insert $\boxed{1}$, in I_{dce} we conservatively require the existence of an unused timestamp interval $(t_r, f']$ on the left of every write message (e.g. $\boxed{8}$) of the source that has a counterpart at the target.

7.2 Applicability and Limitations

Compiler transformations in theoretical classification. Our simulation can verify all the transformations (on non-atomic accesses) in the following categories, except 5), classified by many prior works (e.g. [23]): 1) trace-preserving transformations, which do not change memory accesses; 2) elimination of redundant reads/writes; 3) reordering; 4) introduction of redundant reads; 5) introduction of redundant writes. We don't support 5) because it is unsound in PS. Also we want to emphasize that the analyses-based optimization algorithms verified in our work can do much more than the transformations in the above categories. For instance, DCE can eliminate not only redundant writes, but also those that are found dead in the program analysis; and it can optimize across specific kinds of atomic accesses.

Optimizations in real-world compilers. CompCert [6] performs the following dataflow-analyses based optimizations: ConstProp, CSE, DCE and register allocation. We have verified that ConstProp, CSE and DCE are correct in PS2.1. Register allocation is already challenging to verify in sequential settings and CompCert only does a posteriori validation.

LLVM roughly performs two classes of dataflow-analyses based optimizations: redundancy elimination, and register promotion. Many redundancy elimination optimizations in LLVM can be viewed as specializations or degenerations of the four algorithms verified in our work. For example, LLVM's "dead store elimination" only eliminates basic-block local redundant writes, while DCE we verified can eliminate dead writes *across* basic blocks. We haven't considered either the *inter-procedural* analyses based optimizations or the *alias-analysis* based register promotion in LLVM.

Limitations of our work and possible future extensions. Mainly, there are two kinds of thread-local optimizations which are sound in PS2.1 but cannot be verified by our simulation. The first one is the optimizations on *atomic* memory accesses, such as fence elimination. We haven't considered them because they are rarely found in mainstream C compilers such as GCC and LLVM. The second one is the introduction of additional writes on *thread-local* memory, such as register allocation & spilling, which allocates thread-local memory locations for pseudo registers and converts some operations on pseudo registers to memory accesses. To verify them, we must extend PS2.1 with thread-local memory.

8 Related Work

Many works proved the correctness of real-world optimization algorithms with program analyses in compilers for sequential programs [6, 20, 21, 24], but there is not much discussion about how to prove the correctness of optimizers for concurrent programs in weak memory models.

Gäher et al. [7] develop Simuliris, a simulation technique that establishes termination preservation (under a fair scheduler) for concurrent program transformations while assuming data race freedom of source programs. Their work is built on Iris [9], which gives the SC semantics [12] to concurrent programs, and proves the correctness of transformations on specific code. By contrast, we consider the weak memory model PS2.1, and verify the correctness of optimization algorithms, not just transformations of specific code.

Jiang et al. [8] develop CASCompCert for correct compilation of data-race-free C-like programs. Their source programs are in the SC semantics, i.e. they do not give weak semantics to the various atomic primitives in C/C++ concurrency [1], e.g. release writes and acquire reads. Moreover, they do not support optimizations that may introduce read-write races, such as LICM we verified. They also forbid optimizations across synchronizations, while we investigate the semantics of atomic accesses and show that some optimizations are still safe across proper atomic accesses.

Ševčík et al. [19] develop CompCertTSO, which compiles ClightTSO programs to the x86-TSO machine. The TSO semantics for the source C programs is very different from (and stronger than) the standard C/C++11 weak semantics, and these source programs cannot be compiled to efficient ARM or Power programs. The proof method in CompCertTSO uses an overly strong simulation relation, requiring the source and target to always generate the same memory accesses, which prevents almost all optimizations on memory accesses, including the ones we verified.

For weak memory models of C-like languages formulated in an axiomatic style (e.g. [1, 3, 4, 11]), where the semantics is defined in terms of axioms about complete executions, people have developed tools to *validate* the code transformations (e.g. [2, 15, 16]): for each pair of source and target programs, the validator checks whether the behaviors are the same. Formally proving the correctness of optimization algorithms in these models is still an open problem.

Many works [11, 13, 17, 18] prove the correctness of *compilation schemes* from high-level concurrent programming languages with weak semantics to mainstream multi-core architectures, such as x86-TSO, ARM and POWER. These compilation schemes map each high-level primitive to a sequence of machine instructions directly and do not involve any code optimizations (such as reordering and elimination of reads/writes). It seems difficult to apply their approaches to verify optimization passes.

Comparison with PSSim. One closely related work is the "official" simulation (called PSSim below) proposed together with the promising semantics [10]. We both support thread-local verification by using invariants to abstract the environment interference. However, the invariant in PSSim is fixed (called I_{pssim} below), while ours can be instantiated differently in verifying different optimizations. Also our use of non-preemptive semantics and the assumption of write-write race freedom give users more freedom to instantiate the invariant. Consequently, the proof for an optimization can be greatly simplified. For instance, to verify ConstProp and CSE, we use I_{id} as the invariant, which is much stronger than I_{pssim} and more convenient to use. For DCE, we use an invariant weaker than I_{pssim}, which allows us to establish the lock-step simulation. With the fixed invariant I_{pssim}, it is unclear whether PSSim is applicable to all these optimizations.

On the other hand, we also have to pay extra prices that are not needed in PSSim. We prove the non-preemptive semantics is equivalent to PS2.1 (in Thm. 4.1). Also our simulation needs to be strong enough to ensure (as a meta-property) the target preserves the write-write race freedom of the source. For this purpose, we introduce the delayed write set $\mathcal{D}$.

Acknowledgments

We thank our shepherd Jeehoon Kang and anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61922039 and 61632005.

References

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Webe. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'11)*. 55–66. https://doi.org/10.1145/1926385.1926394

[2] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. 216–226. https://doi.org/10.1145/2854038.2854051

[3] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. 100–110. https://doi.org/10.1109/CGO.2017.7863732

[4] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. In *Proceedings of the ACM on Programming Languages (POPL '19, Vol. 3)*. 1–28. https://doi.org/10.1145/3290383

[5] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. In *Proceedings of the 42nd annual ACM SIGPLAN conference on Programming Languages Design and Implementation (PLDI '21)*. https://doi.org/10.1145/3453483.3454082

[6] CompCert Developers. 2020. CompCert-3.7. http://compcert.inria.fr/release/compcert-3.7.tgz

[7] 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 (2022), 31 pages. https://doi.org/10.1145/3498689

[8] Hanru Jiang, Hongjin Liang, Siyang Xiang, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. https://doi.org/10.1145/3314221.3314595

[9] 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, Vol. 50. Association for Computing Machinery, 637–650. https://doi.org/10.1145/2775051.2676980

[10] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '17)*. https://doi.org/10.1145/3009837.3009850

[11] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. 618–632. https://doi.org/10.1145/3062341.3062352

[12] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[13] 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 *Proceedings of the 41st annual ACM SIGPLAN conference on Programming Languages Design and Implementation (PLDI '20)*. https://doi.org/10.1145/3385412.3386010

[14] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-guarantee-based Simulation for Verifying Concurrent Program Transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. 455–468. https://doi.org/10.1145/2103621.2103711

[15] Dodds Mike, Batty Mark, and Gotsman Alexey. 2018. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *Programming Languages and Systems*, Amal Ahmed (Ed.). 1027–1055. https://doi.org/10.1007/978-3-319-89884-1_36

[16] 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 *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 187–196. https://doi.org/10.1145/2491956.2491967

[17] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *31st European Conference on Object-Oriented Programming (ECOOP'17, Vol. 74)*. 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.22

[18] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2018. Bridging the gap between programming languages and hardware weak memory models. In *Proceedings of the ACM on Programming Languages (POPL'18, Vol. 3)*. 1–32. https://doi.org/10.1145/3290382

[19] 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 (2013), 1–50. https://doi.org/10.1145/2487241.2487248

[20] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages* 4, 23 (2020), 1–31. https://doi.org/10.1145/3371091

[21] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 275–287. https://doi.org/10.1145/2775051.2676985

[22] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 209–220. https://doi.org/10.1145/2775051.2676995

[23] Jaroslav Ševčík. 2011. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, 306–316. https://doi.org/10.1145/1993498.1993534

[24] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages* 3, 62 (2019), 1–30. https://doi.org/10.1145/3290375

[25] Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics – Technical Appendix and Coq Development. https://plax-lab.github.io/publications/promisingcomp/