Putting Weak Memory in Order via a Promising Intermediate Representation

SUNG-HWAN LEE, Seoul National University, Korea
MINKI CHO, Seoul National University, Korea
ROY MARGALIT, Tel Aviv University, Israel
CHUNG-KIL HUR, Seoul National University, Korea
ORI LAHAV, Tel Aviv University, Israel

We investigate the problem of developing an "in-order" shared-memory concurrency model for languages like C and C++, which executes instructions following their program order, and is thus more amenable to reasoning and verification compared to recent complex proposals with out-of-order execution. We demonstrate that it is possible to fully support non-atomic accesses in an in-order model in a way that validates all compiler optimizations that are performed in single-threaded code (including irrelevant load introduction). The key to doing so is to utilize the distinction between a source model (with catch-fire semantics) and an intermediate representation (IR) model (with undefined value for racy reads) and formally establish the soundness of mapping from source to IR. As for relaxed atomic accesses, an in-order model must forbid load-store reordering. We discuss the rather limited performance impact of this fact and present a pragmatic approach to this problem, which, in the long term, requires a new kind of hardware store instructions for implementing relaxed stores. The source and IR semantics proposed in this paper are based on recent versions of the promising semantics, and the correctness proofs of the mappings from the source to the IR and from the IR to Armv8 are mechanized in Coq. This work is the first to formally relate an in-order source model and an out-of-order IR model with the goal of having an in-order source semantics without any performance overhead for non-atomics.

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

Additional Key Words and Phrases: Relaxed Memory Concurrency; Operational Semantics; Compiler Optimizations; Intermediate Representation

ACM Reference Format:
Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proc. ACM Program. Lang.* 7, PLDI, Article 183 (June 2023), 24 pages. https://doi.org/10.1145/3591297

1 INTRODUCTION

Despite decades of research, finding the right semantics for concurrent shared-memory programs in high-level languages is still considered to be a major open problem [Batty et al. 2015], which prevailing languages like C, C++, and Java have yet to overcome. The technical challenge lies in precisely identifying and balancing the conflicting desiderata of programmers, compilers, and modern multicore architectures. Generally speaking, programmers need a simple semantics that

Authors' addresses: Sung-Hwan Lee, Seoul National University, Korea, sunghwan.lee@sf.snu.ac.kr; Minki Cho, Seoul National University, Korea, minki.cho@sf.snu.ac.kr; Roy Margalit, Tel Aviv University, Israel, roy.margalit@cs.tau.ac.il; Chung-Kil Hur, Seoul National University, Korea, gil.hur@sf.snu.ac.kr; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il.

2475-1421/2023/6-ART183
https://doi.org/10.1145/3591297

allows them to reason about their code; compilers strive to apply program optimizations and to be able to justify their correctness; and hardware aims at efficient non-blocking implementations that only provide rather weak consistency guarantees.

Recent years have shown multiple proposals of shared-memory concurrency models that aim to address this challenge (see, *e.g.*, [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2020; Jeffrey et al. 2022; Kang et al. 2017; Lee et al. 2020; Paviotti et al. 2020]). These models typically focus on *performance*, aiming at a semantics that allows various compiler optimizations and efficient mapping to hardware. In particular, to support *load-store reordering* (of accesses to different addresses), either as a part of a compiler optimization or as a possible result of the hardware's pipeline, all these models employ some sort of out-of-order execution that allows reads to read from future writes. For not sacrificing *programmability*, which typically means that "out-of-thin-air" values should be forbidden and the model should admit well-accepted data-race-freedom (DRF) guarantees [Adve and Hill 1990; Batty et al. 2015; Cho et al. 2021], such models have to restrict their speculation mechanisms in a way in which certain program behaviors have to be justified by the existence of other program behaviors. For instance, the promising semantics by Kang et al. [2017] requires promises of future writes to be justified by *another* thread-local run of the program, and event-structure models, as the one by Chakraborty and Vafeiadis [2019], enforce consistency constraints on a structure that captures *several* runs of the program. This makes these models rather complex to reason about, and, indeed, besides several notable exceptions for particular models (see, *e.g.*, [Abdulla et al. 2021; Svendsen et al. 2018]), existing verification research cannot handle such models.

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

The most intuitive example for an in-order semantics is the well-known model of sequential consistency (SC), where different threads take turns communicating with a single global memory in the form of address-to-value mapping, and every read obtains its value from the last previously executed write to the same address. Nevertheless, various other models, weaker than SC, are still in-order. In particular, RC11 [Lahav et al. 2017], a well-studied declarative model for C/C++ that follows the proposal in [Boehm and Demsky 2014] to forbid cycles in the union of the program order and the reads-from relation, is an in-order model. Verification for RC11-style models has been extensively studied, and multiple techniques have been developed, including program logics [Dang et al. 2020, 2022; Doherty et al. 2019], model checkers and fuzzers [Kokologiannakis et al. 2017, 2019; Luo and Demsky 2021], automatic robustness analyses [Margalit and Lahav 2021], and library abstraction theorems [Raad et al. 2019; Singh and Lahav 2023].

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

We target C/C++ as a source language [Batty et al. 2011; Boehm and Adve 2008]. Most importantly, this means that programmers distinguish between synchronization accesses ("atomics") and weak accesses that should not be used for inter-thread synchronization ("non-atomics"), and can cause any behavior when they are misused for this purpose nonetheless. The latter allows us to rely on "undefined behavior" for racy non-atomics, which is a crucial ingredient of our proposed approach. (Thus, we do not provide a solution for "safe" languages that cannot tolerate undefined behavior.)

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

(a) Before optimization

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

  return sum; }
```

(b) After optimization

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

For atomics, we support the main shared memory constructs of C/C++11, including relaxed and release/acquire accesses, read-modify-writes, and release/acquire and sequentially consistent fences.

Non-atomic accesses account for the vast majority of memory accesses in concurrent programs, while atomics, which are used for inter-thread communication and synchronization, are relatively rare. In particular, among atomics, the only ones that are intended to allow the problematic load-store reordering are relaxed accesses, which are meant to be used by "very careful" programmers [Boehm and Adve 2008] and are often confined to libraries that are manually optimized by experts. Thus, we believe that the trade-off between performance and amenability to reasoning should be investigated differently for atomics and non-atomics. Next, we separately discuss the performance overhead that is imposed by an in-order semantics for supporting non-atomic accesses and atomic accesses.

Overhead in Non-atomic Accesses. Our first question is whether it is possible to have an in-order semantics without imposing any performance overhead for non-atomic accesses. This stems from a principled approach: being non-racy, non-atomics should allow all compiler optimizations that are performed in single-threaded code.[1] We observe that a significant challenge exists for validating this guiding principle in an in-order model, and we are not aware of any existing model that solves this challenge (even for a simple fragment with only non-atomics and strong synchronization accesses with, say, release/acquire semantics). In particular, RC11 invalidates (irrelevant) *load introduction*, a transformation widely used in sequential code with significant possible performance gains. In fact, the LLVM manual requires that non-atomics should validate all optimizations allowed on sequential accesses (the only exception is store introduction, which compilers avoid also in sequential code), and explicitly mentions that load introduction may be performed by the compiler, and the LLVM compiler indeed introduces non-atomic loads as a part in several of its optimization passes.[2] The assumptions of the GCC compiler are less clear, but some examples show that it introduces loads as well. A concrete example is given in Fig. 1.

We provide a full solution to this challenge, and design an in-order semantics that does not sacrifice any optimization on non-atomics. Inspired by LLVM, the key to doing so is to utilize the distinction between a *source* semantics and an *intermediate representation* (IR) semantics. This allows the separation of concerns: compiler optimizations may be unsound in the source semantics, whereas the IR semantics does not have to be in-order. Indeed, the IR is not meant to be amenable to conventional verification and reasoning, and programmers in the source language only need to know the source semantics. This strategy, however, is not a magic potion: to have a sound

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

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

compilation, these models have to be designed in a way that the IR semantics is stronger than the source semantics (*i.e.*, all behaviors allowed by the IR should be allowed by the source).[3]

Our main contribution is to show that this approach works with the right choice of source and IR. Concretely, we develop an in-order source model, based on the *promise-free* fragment of the promising semantics, and an IR model based on a recent version of the promising semantics in [Cho et al. 2022], and prove the required relation between them. Our proposed source model is (slightly) stronger than RC11, which allows the application of previous work on verification under RC11. (In particular, we observe that certain races on non-atomics can be safely ignored in RC11's catch-fire mechanism.) For the IR, we have ported the result of [Cho et al. 2022], which establishes the correctness of all optimizations on non-atomics that are allowed in sequential code. This means that most compiler optimizations can be formally validated based on sequential reasoning, so even most compiler developers need not understand the out-of-order IR model.

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

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

What is the practical impact of forbidding this reordering? First, we note that although compiler optimizations that reorder and eliminate atomics were extensively studied before (see, *e.g.*, [Dodds et al. 2018; Vafeiadis et al. 2015]), to the best of our knowledge, existing compilers do not perform any of these optimizations. Then, it remains to understand the implications on the mapping to hardware. Indeed, load-to-store ordering between plain accesses is not guaranteed to be preserved by existing models of modern architectures, like those of Arm [Alglave et al. 2021; Pulte et al. 2017] and Power [Alglave et al. 2014; Sarkar et al. 2011],[4] and so, forbidding this reordering seems to require a stronger mapping of relaxed accesses for these architectures.

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

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

[4]Intel's architecture (assuming x86-TSO by Owens et al. [2009]), has rather strong semantics for plain loads and stores, which never reorders loads with later stores.

[5]Load-store reordering (concretely, the weak behavior of the LB litmus test) was never observed on Power as well as on various implementations of Armv8 that were tested in [Alglave et al. 2021, 2014]. An anonymous review of this paper provided information showing that this reordering is observed on Cortex A73, and mentioned that even on Cortex post A76 load-store reordering can be observed when memory locations are mapped to device, or when vector instructions are used.

Accordingly, we propose a practical approach to this challenge. In the long term, we believe the right way to go is for vendors to introduce new kinds of store instructions, which we call "strong stores", and officially preserve the order from loads to strong stores. We expect a minimal (to no) overhead for these instructions compared to plain stores. In particular, strong stores still admit store-store reorderings, which are commonly observed in practice, and are thus weaker than release stores that are more expensive to implement. Meanwhile, in the absence of such instructions, we propose to compile relaxed writes differently depending on the target hardware: (1) for a target that preserves load-store order, the compilation can use plain accesses; and (2) otherwise, relaxed writes have to be compiled as release writes.

Outline. The rest of this paper is structured as follows. In §2, we present the challenges, key ideas, and observations of this paper in more detail. In §3, we present (a simplified fragment of) the proposed source model and discuss its relation to RC11. In §4, we present (a simplified fragment of) the IR model and establish the soundness of mapping the source model to the IR model. In §5, we discuss the mapping to modern hardware, its soundness, and the proposed additions to hardware models. Finally, in §6, we discuss related work.

Supplementary Material. Our main results ((1) soundness of mapping from source to IR, (2) soundness of mapping from IR to ARMv8, (3) DRF guarantees for the source, and (4) adequacy of sequential reasoning for validating optimizations in the IR) are **mechanized in Coq**. The supplementary material available online [Lee et al. 2023] includes the Coq development, the full models, a (pen-and-paper) proof of the relation to RC11, and the results of our experiments.

2 CHALLENGES AND KEY IDEAS

In this section, we present more details on the main observations and contributions of this paper. To a significant extent, our central contributions are not in developing new concurrency models and proving their meta-theoretic properties but rather in providing a holistic analysis and approach to the problem of a shared-memory concurrency semantics in a high-level language like C, C++, or Rust. Like in §1, we separately discuss non-atomics (§2.1) and atomics (§2.2) while focusing on compiler optimizations for non-atomics and mapping to hardware for atomics. (Our results include the mapping of non-atomics to plain accesses on hardware, as well as compiler optimizations involving atomics, but these are not discussed in this section.)

2.1 Optimizing Non-Atomics in an In-Order Semantics

Supporting sequential optimizations for non-atomics in an in-order semantics is highly challenging, and, to the best of our knowledge, it was not addressed by previous work. Next, we demonstrate the challenge using the well-known load buffering example (§2.1.1); explain how the "catch-fire" addresses this challenge (§2.1.2); describe why catch-fire semantics cannot support load introduction by the compiler (§2.1.3); outline "undefined value" as an (informal) alternative to "catch-fire" and why it fails in combination with an in-order model (§2.1.4); and conclude with our proposal of having a two-layered model with catch-fire source semantics, and undefined-value-based semantics for the intermediate representation (§2.1.5).

2.1.1 ***Read-Write Reordering vs. In-Order Semantics.*** To understand the crux of the challenge, consider the classical example on the right, known as the load buffering litmus test (LB, for short), where all accesses are marked as non-atomics (na). Here and henceforth, we assume that all variables are implicitly initialized to 0. Our requirement on compiler

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

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

2.1.2 ***Catch-Fire as a Solution?*** A well-known approach to address the above example is to exploit the fact that non-atomics are not supposed to be used for inter-thread synchronization and avoid providing any guarantees on the program behaviors when non-atomics participate in data races. This idea, which we refer to as "catch-fire" semantics, is the cornerstone of the C/C++11 [Batty et al. 2011], and its repaired version RC11 [Lahav et al. 2017], which explicitly states that a data race on non-atomic accesses implies *undefined behavior* (UB, for short) for the given program.

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

Remark 1. The original presentation of RC11 in [Lahav et al. 2017] identifies program behaviors with "final outcomes" (mapping each variable to the modification-order-maximal value written to it). The current discussion assumes that behaviors are captured by sequences of system calls (*e.g.*, results of print statements) generated by a given program. RC11 can be easily adapted to this notion by assuming that consistent execution graphs are incrementally constructed during the program run, system calls are observed in the order they were executed along the run, and any suffix of system calls is allowed once a racy execution graph is reached.

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

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

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

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

Load introduction is necessary for multiple optimizations based on speculation, which are commonly performed by compilers (Clang, in particular) when hoisting loads, *e.g.*, as a part of loop invariant code motion, loop unswitching, load-widening or when loading a vector while only a subset of elements is needed.[6] In addition to Fig. 1 from §1, Fig. 2 demonstrates another case where load introduction has the potential to significantly improve performance.

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

```
1  extern void foo(char* x), bar(char* x);
2  int main() {
3    char x[8], y[8];
4    for (int i = 0; i < 10000000; i++) {
5      foo(x);
6      for (int j = 0; j < 8; j++)
7        y[j] = j%2 ? x[j] : 0;
8      bar(y); }
9    return 0; }
```

(a) Before optimization

```
1  extern void foo(char* x), bar(char* x);
2  int main() {
3    char x[8], y[8];
4    for (int i = 0; i < 10000000; i++) {
5      foo(x);
6      uint64_t r = *(uint64_t*)x;
7      *(uint64_t*)y = r & 0xFF00FF00FF00FF00ul;
8      bar(y); }
9    return 0; }
```

(b) After optimization

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

2.1.4 ***Undefined Value as a Solution?*** A natural idea for supporting load introduction is to limit the "undefinedness" to the value being read in racy reads: instead of invoking UB, just leave unspecified the value loaded by a non-atomic racy read, so if this value is never used (and the load is indeed irrelevant), we will not introduce additional behaviors. The LLVM semantics follows this idea: it keeps read-write races to be always well-defined and declares that non-atomic racy reads may return "undef" value. In turn, "undef" can be refined to *any* value.[7]

While being tempting at first sight, undefined value for racy reads will not solve our problem. Referring back to the LB example above, it is easy to see that any execution of an in-order semantics can observe a race only in one of the reads, so only one of them can return "undef", which will not allow *both* threads to print 1. To fix this, one has to either speculate a data race when performing the first read, or revisit its previous decisions on the read value when performing the second write. Both options lead us to models that are much more complicated than in-order models.

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

(1) We propose a source model, which we denote by vRC11, obtained by adding non-atomic accesses to the *promise-free* fragment of the promising semantics [Kang et al. 2017; Lee et al. 2020]. This model, which is stronger than RC11, is formulated as an operational model using timestamps and thread-views to justify weak behaviors, and a simple race-detection mechanism that invokes UB for races on non-atomics. Interestingly, we observe that not all such races should invoke UB, and it is sufficient to consider races with previously executed writes (and ignore races with previously executed reads). Thus, we are able to restrict the catch-fire mechanism in a way that deems fewer programs racy but still achieves what catch-fire is needed for.

[7]Branching on "undef" is still considered UB. The "freeze" instruction recently introduced in LLVM is a tool to support branching on a possibly undefined value, which is often a result of load introduction [Lee et al. 2017].

(2) For the IR model, we develop a simplification of the promising model by Cho et al. [2022], which we denote by PS^{IR}, where (simplified) promises are only needed for race detection. Thus, PS^{IR} justifies an out-of-order behavior by detecting a race with "promises" made by other threads. More concretely, a thread in PS^{IR} can *promise* to execute a non-atomic write to a location X in the future, whenever the thread can *certify* the promise by checking that it can perform a non-atomic write to X by executing alone. Once a promise is made, another thread reading from X races with the promise and reads "undef" value. We have ported the result of [Cho et al. 2022] to PS^{IR}, which establishes the correctness of all optimizations on non-atomics that are allowed in sequential code.

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

Example 2.1. In the LB example, PS^{IR} allows $a = b = \texttt{undef}$ through an out-of-order execution where the first thread promises to write to Y, and the second thread reads "undef" from Y since the read races with the promise. The first thread could certify its promise before making it by reading 0 from X and executing $Y^{\texttt{na}} := 1$. In vRC11, instead of promising the write, the first thread can execute and write to Y following the certification execution of PS^{IR}. Then, the second thread's read from Y becomes racy, and the program invokes UB, which accounts for all possible behaviors.

Remark 2. In fact, since UB by the source accounts for any behavior of PS^{IR}, the proof of mapping the source to the IR can essentially assume that there is no race in the promise-free fragment of PS^{IR}, which makes the mapping proof similar to the proof of the DRF-PF theorem (a data-race-freedom guarantee w.r.t. the promise-free semantics) in [Cho et al. 2021].

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

2.2 Mapping Relaxed Accesses to Modern Hardware

In this section, we turn to the question of supporting atomic accesses focusing specifically on relaxed accesses. We demonstrate the challenge (§2.2.1); revisit the assumptions on hardware (§2.2.2); and propose two practical solutions: a long-term solution that depends on hardware vendors implementing our feature request (§2.2.3), and a short-term solution that requires strengthening the existing compiler mapping of relaxed accesses (§2.2.4).

2.2.1 *Reordering of Relaxed Accesses in an In-Order Semantics.*

Like non-atomic accesses, relaxed accesses in C/C++11 were intended to be mapped to plain loads and stores in the hardware even when the hardware model allows load-store reordering. Clearly, this is in contrast with an in-order semantics (indeed, consider the LB example above with relaxed accesses). Moreover, since relaxed accesses are meant to be used in races (for improving the performance of certain concurrency idioms; see, *e.g.*, [Sinclair et al. 2017]), catch-fire is not a possible solution here. In fact, as the next example shows, even the reordering of a non-atomic load followed by a relaxed atomic

(1)	(2)	(3)	(4)	(5)	(6)
$c := X^{\mathsf{na}}$	$c := X^{\mathsf{na}}$	$c := X^{\mathsf{na}}$	$c := X^{\mathsf{na}}$	$c := X^{\mathsf{na}}$	$Y^{\mathsf{rlx}} := 1$
				$Y^{\mathsf{rlx}} := 1$	$c := X^{\mathsf{na}}$
`if` $*$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`	`if` $c = 1$ `then`
$b := X^{\mathsf{na}}$	$b := X^{\mathsf{na}}$	$b := 1$	$b := 1$		
`if` $b = 1$ `then`	`if` $b = 1$ `then`	`if` $b = 1$ `then`			
$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$		
`print` b	`print` b	`print` b	`print` 1	`print` 1	`print` 1
`else`	`else`	`else`	`else`		
$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$	$Y^{\mathsf{rlx}} := 1$		

(1) introduce a non-atomic read $c := X^{\mathsf{na}}$; (2) replace the non-deterministic choice with an expression; (3) forward the read $c := X^{\mathsf{na}}$ to the read $b := X^{\mathsf{na}}$ in the if-branch, turning it into $b := 1$; (4) forward $b := 1$ to the expression $b = 1$ and the print statement; (5) hoist the common write $Y^{\mathsf{rlx}} := 1$ out of the branch; and (6) reorder $c := X^{\mathsf{na}}$ and $Y^{\mathsf{rlx}} := 1$.

Fig. 3. A sequence of compiler transformations on non-atomics (1–5) and the problematic reordering of a non-atomic load followed by a relaxed store (6) applied to the second thread of LB-CHOICE.

store cannot be allowed in an in-order semantics:

$$\begin{array}{l} a := Y^{\mathsf{rlx}} \\ \texttt{if } a = 1 \texttt{ then} \\ \quad X^{\mathsf{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{if } * \texttt{ then} \\ \quad b := X^{\mathsf{na}} \\ \quad \texttt{if } b = 1 \texttt{ then } Y^{\mathsf{rlx}} := 1 \\ \quad \texttt{print } b \;\; \textit{// prints 1} \\ \texttt{else} \\ \quad Y^{\mathsf{rlx}} := 1 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := Y^{\mathsf{rlx}} \\ \texttt{if } a = 1 \texttt{ then} \\ \quad X^{\mathsf{rlx}} := 1 \end{array} \;\Big\|\; \begin{array}{l} Y^{\mathsf{rlx}} := 1 \\ c := X^{\mathsf{na}} \\ \texttt{if } c = 1 \texttt{ then} \\ \quad \texttt{print } 1 \end{array} \qquad \text{(LB-CHOICE)}$$

Here, "$*$" means a *non-deterministic choice* that non-deterministically returns arbitrary value.[8] Assuming an in-order semantics, the source program on the left cannot print 1 since either one of $a := Y^{\mathsf{rlx}}$ or $b := X^{\mathsf{na}}$ executes first and can only read 0 (from the initial memory). However, as shown in Fig. 3, by applying a sequence of compiler transformations on non-atomics and finally reordering (by the compiler or the target hardware) $c := X^{\mathsf{na}}$ and $Y^{\mathsf{rlx}} := 1$ in the second thread, the program on the left can be transformed into the program on the right. Then, the second thread printing 1 is easily observable (even under SC). Therefore, the reordering of a non-atomic load followed by a relaxed store must be *forbidden* in any in-order source semantics that aims to allow common compiler transformations on non-atomics.

2.2.2 ***Revisiting the Assumptions on Hardware.*** We observe that there is a significant gap between CPU *models* and observable behaviors *in practice* regarding the preservation of load-store ordering. While the abstract models effectively allow the reordering of loads with subsequent stores, such behaviors are rarely observed in practice. Indeed, previous experiments performed to validate the hardware models rarely observed weak behaviors of the LB litmus test. First, such behaviors were never observed on any Power hardware [Alglave et al. 2014; Sarkar et al. 2011]. Second, while they were observed on several Armv7 implementations, to the best of our knowledge, for Armv8, LB was only observed on Qualcomm's Snapdragon 820 mobile processors [Alglave et al. 2021] and on Cortex A73.[9] To gain more confidence, we experimentally tested a newer version, Qualcomm's Snapdragon 888 processor, and the weak behaviors of LB were not observed there.

After discussing with Arm engineers, we gained a better understanding of the architectural reasons why the potential performance improvement by allowing load-store reordering is relatively

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

[9] Snapdragon 820 exhibits various other weak behaviors that are forbidden by the official model (950 such tests reported in [Alglave et al. 2021]!). The information about Cortex A73 was obtained from the anonymous PLDI reviewer.

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

Remark 3. Unlike load-store ordering, preserving store-store ordering is rather expensive. For instance, in Cortex A76 and later versions, a store from the store buffer is committed to a *merge buffer* when it is the oldest store (*i.e.*, all preceding loads are completed, and all preceding stores are already committed to the merge buffer). Then, stores in the merge buffer may be reordered to group together those writes that fit in the same cache line, which are merged and committed at once. Such reordering between stores greatly reduces cache accesses and is thus considered performance-critical, which is why store-store reordering visible to other cores is needed.

2.2.3 ***A Long Term Practical Solution.*** Based on the above discussion, we raise a clear "feature request" from hardware vendors. Concretely, we propose hardware vendors to introduce a new kind of store instructions, which we call "strong stores", that will preserve load-store ordering. Then, the IR's relaxed stores will be mapped to strong hardware stores. For most hardware architectures, where architects agree that load-store ordering is preserved, strong stores could be implemented as plain stores. Otherwise, the overhead is not expected to be significant, and, in any case, strong stores should be cheaper than release stores (since they do not need to preserve store-store order).

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

In §5, we provide the proposed formal additions to the declarative models of Armv8 and Power for supporting strong stores. We have performed extensive validation of these revised models using the Herd model checker [Alglave et al. 2021, 2014], to see that, indeed, when strengthening all stores to be strong, the behaviors that become disallowed are, like LB, behaviors that were not observed on hardware (except for Snapdragon 820 and Cortex A73 as discussed above).

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

Following Ou and Demsky [2018], mapping relaxed stores as release entails a performance overhead of 3.6% on Arm (although it is rather hard to estimate performance for real-world programs). We note that the mapping scheme that enforces the preservation of load-store ordering by inserting a (fake) branch from every relaxed read, which is more efficient according to Ou and Demsky [2018] (with -0.3% overhead), is *unsound* for our needs. Indeed, as the LB-CHOICE example shows, we also need to forbid reordering of *non-atomic* reads followed by relaxed writes. This would require adding a branch from every non-atomic read, which, given the prevalence of non-atomic reads in concurrent programs, is expected to significantly harm performance.

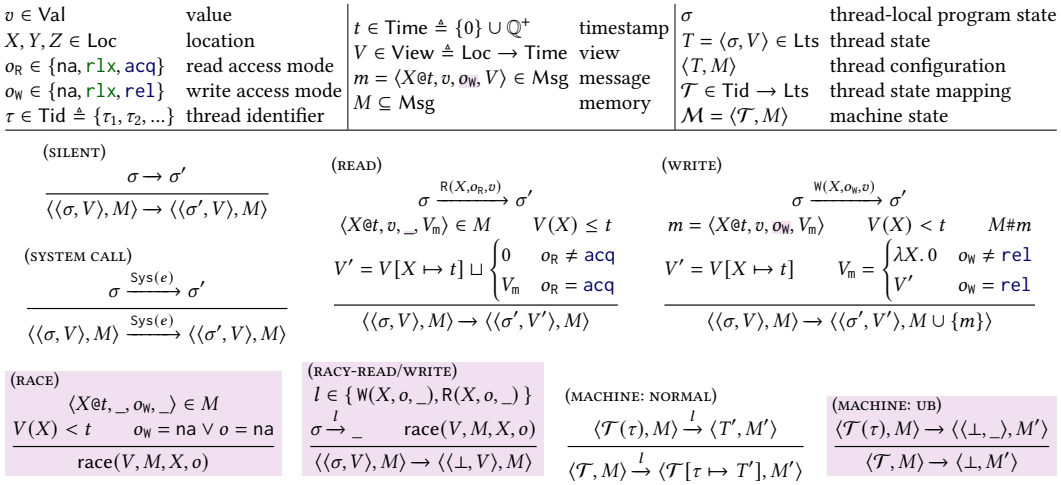


Fig. 4. Domains and transitions of vRC11 (RMWs, fences, and release sequences are omitted). Differences w.r.t. the promise-free fragment of PS are highlighted.

3 THE SOURCE MODEL

In this section, we present the in-order source semantics vRC11 (standing for "view-based RC11"), which we obtain by adding transitions for non-atomic accesses to the promise-free fragment of the promising semantics (PS, for short). In §3.1, we discuss the relation between vRC11 and RC11 and show that vRC11 is stronger than RC11. Therefore, verification theory and tools developed for RC11 (or any weaker model), such as model checkers [Kokologiannakis et al. 2017, 2019; Luo and Demsky 2021], program logics [Dang et al. 2020; Doko and Vafeiadis 2017], and robustness analysis [Lahav and Margalit 2019], all apply to vRC11. In §3.2, we provide a declarative presentation of the model.

vRC11 is obtained from PS by (i) removing the notion of *promises* that models early execution of writes and all transitions and components of states related to promises; and (ii) adding transitions for non-atomic and racy accesses. Next, we introduce the fragment of vRC11 consisting of non-atomic, relaxed and release/acquire writes and reads. In turn, read-modify-writes (RMWs), fences, and release sequences are omitted by brevity. They are included in the full model in Coq and presented in [Lee et al. 2023, Appendix A]. Figure 4 summarizes the domains and the transitions of vRC11, highlighting the differences w.r.t. the promise-free fragment of the model in [Kang et al. 2017].

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

Memory. A *memory* M is a finite set of *messages* of the form $m = \langle X@t, v, o_\mathsf{W}, V_\mathsf{m}\rangle$ representing a previously executed write of a value $v \in \mathsf{Val}$ to a location $X \in \mathsf{Loc}$. Each message has a *timestamp* $t \in \mathsf{Time}$, where Time is the set of non-negative rational numbers,[10] a *write access mode* o_W of the

[10] As in previous work [Kang et al. 2017; Lee et al. 2020], timestamps are densely ordered, so one can always add a message between existing messages. This property is particularly useful when proving the soundness of compiler transformations such as "store merge" that merges two successive stores $X := 1\,; X := 2$ into a single store $X := 2$. In the proof, the source has to mimic the target program by finding a free timestamp for $X := 1$ before $X := 2$.

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

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

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

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

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

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

Behavior. An (observable) *behavior* is a sequence $s = \langle e_1, e_2, \ldots, e_n\rangle$ of system calls. A machine state $\mathcal{M}$ *generates* a behavior s, denoted by $\mathcal{M} \Downarrow s$, if s is obtained by restricting a trace of vRC11 starting from $\mathcal{M}$ to system call labels and replacing UB by an arbitrary suffix of system calls. With standard notations for sequences, $\mathcal{M} \Downarrow s$ is defined by:

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

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

Example 3.1. The "store buffering" test on the right demonstrates how the memory and the thread views of vRC11 captures weak behaviors exhibited by the reordering of a store followed by a load. The behavior of both threads printing 0 is allowed by vRC11.

$$\begin{array}{l|l} X^{\mathsf{rlx}} := 1 & Y^{\mathsf{rlx}} := 1 \\ a := Y^{\mathsf{rlx}} & b := X^{\mathsf{rlx}} \\ \mathtt{print}\ a & \mathtt{print}\ b \end{array} \qquad \text{(SB)}$$

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

Example 3.2. We show how vRC11 allows both threads printing 1 in the LB example in §2. Suppose that the first thread reads 0 from the initial message for X, and writes 1 to Y by adding

a message $m_Y = \langle Y@t, 1, \mathsf{na}, \lambda X.\, 0\rangle$ with some $t > 0$. Then, the read from Y by the second thread races with the message m_Y since it has a timestamp t greater than the timestamp of Y in the second thread's view (*i.e.*, $V(Y) = 0 < t$). Then, due to the racy read from Y, the second thread invokes UB, which generates arbitrary behavior, including the behavior in which both threads print 1.

Example 3.3. Consider the message passing program on the right. The two non-atomic accesses to the data D are well-synchronized by a release-acquire synchronization through the flag F, and thus, they are not racy. Indeed, the first thread records its view in the message $F = 1$ and the view is transferred to the second thread when it reads $F = 1$. The read from D by the second thread is not racy since the timestamp of D it has in its view is already increased to include the timestamp of the message $D = 42$. Moreover, the second thread is only allowed to read 42 from D. In contrast, the program becomes racy if any (or both) of the accesses to F is made relaxed. Then, there would not be a release-acquire synchronization between the two threads, and the timestamp of D in the second thread's view would remain 0 (pointing to the initial message of D) even after reading 1 from F. In turn, the read from D would be racy and invoke a UB, as the message $D = 42$ would have a higher timestamp than the second thread's view of D.

$$\begin{array}{l|\!|l} D^{\mathsf{na}} := 42 & a := F^{\mathsf{acq}} \\ F^{\mathsf{rel}} := 1 & \texttt{if } a = 1 \texttt{ then} \\ & \quad b := D^{\mathsf{na}} \end{array} \qquad \text{(MP)}$$

We have ported the Coq proof by Cho et al. [2022] to establish the *local DRF guarantees*, LDRF-RA and LDRF-SC, for vRC11. Generally speaking, data-race-freedom (DRF) guarantees ensure "strong" semantics for programs that are race-free under the "strong" semantics, and thus provide an essential formal justification for defensive programming. *Local* DRF (LDRF) guarantees further extend this idea to be applicable also in the presence of races on some unrelated locations (*e.g.*, confined in optimized libraries). LDRF-*RA* means that we consider release/acquire semantics as the strong semantics, and LDRF-*SC* means that under the strong semantics, threads can only access messages with globally maximal timestamps.

3.1 Relating vRC11 to RC11

The RC11 [Lahav et al. 2017] memory model addresses two problems of the C/C++11 model: its flawed semantics for sequentially consistent accesses and fences (which is unrelated to the current paper) and the more crucial problem of "out-of-thin-air" reads [Batty et al. 2015] that breaks the fundamental DRF guarantee. To solve the latter problem, following [Boehm and Demsky 2014], RC11 takes a conservative approach and forbids cycles in the union of the program order and the reads-from relation. As discussed before, verification of concurrent programs under RC11 has been extensively studied and multiple verification methods and tools have been developed. The next theorem states that vRC11, the source model of the present paper, is stronger than RC11. Hence, the soundness of all verification approaches for RC11 applies to vRC11 as well.

Theorem 3.4. *For every program prog,* $[\![prog]\!]_{\text{vRC11}} \subseteq [\![prog]\!]_{\text{RC11}}$.

We provide a (pen-and-paper) proof in [Lee et al. 2023, Appendix C], based on a declarative presentation of vRC11 (see §3.2), which can be more easily compared to RC11. Next, we demonstrate behaviors allowed by RC11 but disallowed by vRC11 using examples.

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

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

In both vRC11 and RC11, the read of X has to return 0, but this program is considered racy in RC11 but not in vRC11. Specifically, both vRC11 and RC11 allow the execution where the second thread

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

Remark 4. Deeming programs like RW-RACE as non-racy may allow performance improvements in certain programming idioms that are forbidden in RC11. For example, consider the following multiple-readers-single-writer (MRSW) lock pattern:

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

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

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

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

In both vRC11 and RC11, the read of X has to return 1, but, again, this program is racy in RC11 but not in vRC11. To see this, consider an execution where the relaxed read $a := X^{\mathtt{rlx}}$ by the second thread reads 1 written by the first thread. The write $X^{\mathtt{rlx}} := 1$ by the first thread and the non-atomic read $b := X^{\mathsf{na}}$ by the second are racy in RC11 since they are not well-synchronized via a release-acquire synchronization. In vRC11, the two accesses are not racy: once the second thread reads 1 by the relaxed read $a := X^{\mathtt{rlx}}$, its view to X increases to include the message $X = 1$. Then, when the thread performs a non-atomic read from X, no message to X has a timestamp higher than the thread's view (*i.e.*, only the message $X = 1$ can be read by the second thread). Therefore, the non-atomic read by the second thread does not race with the write of the first thread.

Both of the above examples demonstrate cases that RC11 assigns UB to a program, whereas vRC11 gives it a defined semantics. We believe that races in vRC11 have a clear and simple meaning: a read access is racy iff it can read from more than one message, and a write access is racy iff it

can "overwrite" more than one message. The examples above show cases where RC11 forces a non-atomic read to read from a particular write, but the read is still considered racy in RC11.

Finally, there also is a difference between the two models related to SC-fences (which are not presented above but included in the full model). In vRC11 the semantics of SC-fences is similar to the one in the RC20 model in [Margalit and Lahav 2021], which is stronger than their semantics in RC11. In particular, SC-fences in vRC11 model can be expressed in terms of a release and acquire fences and an RMW to an otherwise unused location (see [Margalit and Lahav 2021, Remark 1]). We do not discuss further this difference since it is orthogonal to our main topic.

3.2 A Declarative Presentation

For informed readers, we provide a declarative (a.k.a. axiomatic) presentation of vRC11. Such presentation is more concise than the operational one, and it is especially useful for comparing vRC11 to other models that are presented in a similar declarative fashion such as C/C++11. In the following, we consider the full model with RMWs and fences. Due to lack of space, we refer to [Lahav et al. 2017], which we build on, for more background and examples of this definition style. (In any case, this technical section can be skipped when reading the paper.) The equivalence between the operational and declarative models is proved in [Lee et al. 2023, Appendix B].

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

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

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

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

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

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

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

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

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

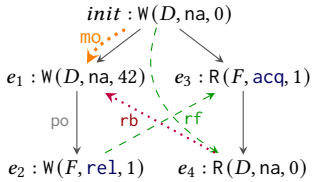
Then, consistent graphs are defined as follows.

Definition 3.5. An execution graph G is vRC11-*consistent* if the following hold for its relations:

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

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

Example 3.6. As we saw in Example 3.3, in the MP program the second thread can only read 42 from D. To see how this follows from the declarative model, we depict the execution graph obtained when trying to read 0 (the initial value), and explain why it is inconsistent. The nodes, labels, and program order (po) arise from the program behavior we analyze. Then, the reads-from relation (rf) is forced since every read has to read its value from some write writing that value. The modification order (mo) between *init* and e_1 is also forced: it has to order these two nodes (as both write to the same location), and going in the opposite order would violate COHERENCE as we have hb from *init* to e_1. Then, according to the definition above, a "reads-before" edge (rb) is induced from e_4 to e_1, which implies $\langle e_4, e_1\rangle \in \mathtt{eco}$. Now, since e_2 and e_3 are rel and acq, we have an sw edge between them, inducing hb from e_1 to e_4. Together, this violates COHERENCE since we have $\langle e_1, e_1\rangle \in \mathtt{hb}\,;\mathtt{eco}$.



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

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

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

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

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

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

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

Remark 5. An equivalent model is obtained if we include sc inside hb (together with po and sw). In this presentation, SC-FENCE is not needed, and the definition of pb can be simplified to $\mathtt{rf}^?\,;\mathtt{hb}$.

Example 3.7. The RW-RACE program above does not generate a racy vRC11-consistent execution graph. Indeed, the only vRC11-consistent execution graph of it executing both non-atomic

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

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

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

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

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

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

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

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

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

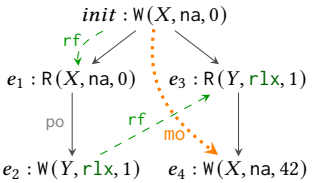
(SYSTEM CALL)
$$\frac{\sigma \xrightarrow{\mathsf{Sys}(e)} \sigma' \quad P = \emptyset}{\langle \langle \sigma, V, P \rangle, P_G, M \rangle \xrightarrow{\mathsf{Sys}(e)} \langle \langle \sigma', V, P \rangle, P_G, M \rangle}$$

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

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

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

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



4 THE IR MODEL

In this section, we present our model of the intermediate representation, called $\mathrm{PS}^{\mathrm{IR}}$, establish the soundness of mapping from vRC11 to $\mathrm{PS}^{\mathrm{IR}}$, and show that every sound transformation under the sequential semantics SEQ in [Cho et al. 2022] is also sound under $\mathrm{PS}^{\mathrm{IR}}$.

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

In the following, we explain the $\mathrm{PS}^{\mathrm{IR}}$ model focusing on how it extends and modifies vRC11. Figure 5 summarizes the steps of $\mathrm{PS}^{\mathrm{IR}}$. We note that the (SILENT) and (READ) steps are adapted in an obvious way that does not alter the new components of the state, and the (RACE) definition is also exactly as in vRC11. The full $\mathrm{PS}^{\mathrm{IR}}$ model in our Coq development includes RMWs, fences, and release sequences, and is presented in [Lee et al. 2023, Appendix D].

Promises. Both each thread state and the global machine state are equipped with a set of locations, called *local promises* (P) and *global promises* (P_G), respectively. The (PROMISE) step allows a thread to *promise* to write to a certain location X in the future by adding X both to its local promises and to the global promises, provided that X has not been already promised by some

thread ($X \notin P_G$). Once X being promised, the thread can later *fulfill* its promise by writing to X via a non-atomic write. Accordingly, the (WRITE) step is extended to update the local and global promises set by removing X from them if the write is non-atomic and X was previously promised.

Certification. At each machine step (see (MACHINE: NORMAL)), the thread taking a sequence of steps should *certify* its promises by demonstrating it is able to fulfill all its promises by taking multiple steps in isolation. The certification requirement is crucial in proving the soundness of mapping from vRC11 to PS$^{\mathtt{IR}}$ (see Example 4.4 below).

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

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

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

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

Example 4.1. In contrast to vRC11, in which a racy read invokes UB, in PS$^{\mathtt{IR}}$ a racy read returns `undef`. Thus, PS$^{\mathtt{IR}}$ justifies LB example using a promise. Specifically, the first thread promises to Y, certified by reading 0 from X and writing 1 to Y. Then, the read from Y by the second thread is racy with that promise and it returns `undef`. After the racy read, the second thread writes 1 to X and prints 1. (Since `undef` represents an arbitrary value, it can be refined to 1.) Now, the first thread reads 1 from X, fulfills its promise to Y by performing a non-atomic write, and prints 1 as well.

Example 4.2. The following example demonstrates that a promise can be certified and fulfilled by different write instructions even with different written values.

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

The program on the left can be transformed into the program on the right by applying a sequence of compiler transformations in the second thread: (*i*) split the non-atomic write $X^{\mathsf{na}} := 1$ into two writes $X^{\mathsf{na}} := 2$ followed by $X^{\mathsf{na}} := 1$; (*ii*) hoist the common write $X^{\mathsf{na}} := 2$ out of the branch; and (*iii*) reorder the read $b := Y^{\mathtt{rlx}}$ and the write $X^{\mathsf{na}} := 2$. Since the program on the right is allowed to print 42 (even under SC), PS$^{\mathtt{IR}}$ should allow the same behavior for the program on the left. Indeed, the program on the left can print 42 through the following PS$^{\mathtt{IR}}$ execution: (1) the second thread promises to X, certifying it by reading 0 from Y and writing 2 to X; (2) the first thread reads `undef` from X by performing a racy read and writes `undef` to Y; and (3) the second thread reads `undef` from Y, enters the then-branch as $b \neq 0$ evaluates to `undef`,[11] fulfills its promise to X by writing 1

[11] Here, we assume that branching on `undef` non-deterministically takes either one of the then-branch or the else-branch. In LLVM, branching on `undef` is UB, and a "freeze" instruction should be used before the branching [Lee et al. 2017].

to X, and prints 42. Notably, in this execution, the promise to X of the second thread is certified using the write $X^{\mathsf{na}} := 2$ and fulfilled by the other write $X^{\mathsf{na}} := 1$.

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

Theorem 4.3. *For any program prog,* $[\![prog]\!]_{\mathrm{PS}^{\mathrm{IR}}} \subseteq [\![prog]\!]_{\mathrm{vRC11}}$.

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

Example 4.4. Theorem 4.3 does not hold without the certification of promises. Under vRC11, the only possible execution for this program is that both threads read the initial messages (*i.e.*, $a = b = 0$). For Thm. 4.3 to hold, $\mathrm{PS}^{\mathrm{IR}}$ cannot allow any other behavior. Indeed, under $\mathrm{PS}^{\mathrm{IR}}$, the first thread cannot promise to Y since the only value that can be read from X is 0, and thus, the thread cannot *certify* the promise. However, if $\mathrm{PS}^{\mathrm{IR}}$ allowed a thread to promise without certifying it, then $a = b = \mathsf{undef}$ would be allowed. Specifically, the first thread could unconditionally promise to Y; the second thread could read `undef` and write 1 to X; and then the first thread could read `undef` and write 1 to Y while fulfilling its promise to Y.

```
a := X^na          ||  b := Y^na
if a = 1 then      ||  if b = 1 then
  Y^na := 1        ||    X^na := 1
```
(LB-DRF)

In addition to Thm. 4.3, we also proved that program transformations sound in sequential semantics are also sound to apply on non-atomics in $\mathrm{PS}^{\mathrm{IR}}$. To do so, we adapted the sequential machine SEQ from [Cho et al. 2022, Def. 3.3] to include non-promisable relaxed writes, as we have in $\mathrm{PS}^{\mathrm{IR}}$ (see [Lee et al. 2023, Appendix E]), and ported the proof in [Cho et al. 2022] to $\mathrm{PS}^{\mathrm{IR}}$ to show that all sound optimizations on non-atomics under (the adapted) SEQ are also sound under $\mathrm{PS}^{\mathrm{IR}}$. Thanks to this result, not only the programmers but also compiler writers who develop optimizations on non-atomic code (including reorderings and eliminations of non-atomic accesses across atomics) do not have to understand the out-of-order IR model.

The sequential machine SEQ, however, is not helpful for validating reorderings and eliminations of *atomics*. These optimizations are not important for our current purpose (to the best of our knowledge, they are not performed by current compilers). In fact, we found out that reordering of relaxed writes (to different locations) is unsound in $\mathrm{PS}^{\mathrm{IR}}$. (Still, $\mathrm{PS}^{\mathrm{IR}}$ can be soundly mapped to Armv8, which effectively allows reordering in the target code; see §5.) The reason is related to the *reservation* mechanism, an addition that was introduced to PS in [Lee et al. 2020] and used in our full $\mathrm{PS}^{\mathrm{IR}}$ model, in order to support an efficient mapping of RMWs to Armv8. Future work is required to understand whether $\mathrm{PS}^{\mathrm{IR}}$ can be changed to allow this reordering. We expect all other transformations on atomics that are sound in RC11 to be sound in $\mathrm{PS}^{\mathrm{IR}}$.

`freeze`(v) returns v when v is a defined value (*i.e.*, not `undef`) and non-deterministically returns any defined value (*e.g.*, 42) when v is `undef`. In the example, the same argument holds when $b \neq 0$ is replaced with `freeze`(b) $\neq 0$.

5 MAPPING TO HARDWARE

In this section, we consider the compiler mapping of PS^{IR} to hardware: we present the proposed addition of "strong stores" to hardware models (§5.1); discuss the implementation of strong stores in existing hardware (§5.2); and establish soundness of mapping PS^{IR} to the extended models (§5.3).

5.1 Strong Stores in Hardware Models

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

Armv8. We define Armv8S as the extension of the Armv8 memory model [Alglave et al. 2021; Pulte et al. 2017] with strong stores. The Armv8 memory model defines a relation called *barrier-ordered-before* (bob), modeling thread-local order of memory accesses that is induced by barriers and release/acquire accesses. For example, bob includes po ; [L] that corresponds to the fact that a release store (denoted by L) is never reordered with an earlier instruction in the program order (denoted by po). In Armv8S, we extend bob to include also [R] ; po ; [S], where S represents the set of strong stores. This simple modification enforces the preservation of the order of any load followed by a strong store in the program order.

Power. Similarly to Armv8S, we define PowerS by extending the Power memory model of [Alglave et al. 2014]. Specifically, we propose a modest extension of the "no-thin-air" rule of the Power consistency predicate that requires acyclicity of ppo ∪ fence ∪ rfe to include [R] ; po ; [S] as well. Roughly, this constraint forbids load buffering behaviors when the order of the load followed by the store is preserved by certain dependencies (ppo) or fences (fence). The PowerS model extends this rule to prevent the load-store reordering also when the order is preserved by a strong store.

5.2 Implementing Strong Stores on Existing Hardware

As discussed in §2.2, the weak behavior of LB has been rarely observed in practice, despite massive testing on CPU implementations of multiple Arm and Power architectures. In particular, among the Armv8 and Power implementations that have been tested in [Alglave et al. 2021, 2014], only Qualcomm's Snapdragon 820 processor exhibited this behavior.

To gain more knowledge about the Snapdragon anomaly (and extend the dataset of [Alglave et al. 2021]), we experimented with a new Snapdragon version. We acquired a Snapdragon 888 (SM8350) processor, and using the Litmus7 (part of DIY7) testing framework, we ran the 23 basic behavior tests.[12] Like other processors and unlike Snapdragon 820, Snapdragon 888 did not exhibit the weak behavior of LB in 6000M runs. For comparison, weak behaviors of the well-known store buffering (SB) and message passing (MP) tests were observed in 93% and 0.676% (respectively) of the 6000M runs. The supplementary material [Lee et al. 2023] includes the full results for Snapdragon 888.

On all those implementations that do not exhibit load buffering, we believe that strong stores could be implemented *just like plain stores*, without any additional overhead. To validate this claim, we used the Herd7 model checker. We started from the available tests in the suite of [Alglave et al. 2021, 2014], which includes 3,773 tests for Armv8[13] and 3,116 tests for Power,[14] and confirmed that all behaviors that are forbidden by the strengthened hardware models where every store is strong (*i.e.*, the models obtained by including [R] ; po ; [W] in the bob relation of Armv8 or the

[12] http://gallium.inria.fr/~maranget/cats7/model-aarch64/tests.html [Accessed November 2022].
[13] https://gallium.inria.fr/~maranget/cats7/model-aarch64/index.html [Accessed November 2022].
[14] https://gallium.inria.fr/~maranget/cats7/ppc9/ [Accessed November 2022].

"no-thin-air" constraint of Power), but allowed by the existing models, were never observed on an implementation (except for Snapdragon 820). The supplementary material [Lee et al. 2023] includes the `cat` files defining the models, the tests we ran, and the logs of the results.

5.3 Mapping PS^{IR} to Hardware

Given strong stores in hardware, the mapping of PS^{IR} to hardware follows the standard schemes of C/C++ concurrency primitives,[15] except that relaxed writes in PS^{IR} are mapped to *strong* stores (while non-atomic writes are compiled to plain stores). We do not assume here that the hardware generally preserves load-store ordering, in which case, strong stores are not needed at all. In addition, the soundness of the "short-term" solution (see §2.2), which, in the absence of strong stores, suggests mapping relaxed writes as if they were release, follows from the discussion below since release writes are mapped to constructs that provide stronger guarantees than strong stores.

Remark 6. As was observed in [Cho et al. 2021], to be able to match every out-of-order execution to an in-order racy execution (which we need for Thm. 4.3, and Cho et al. [2021] need for LDRF-PF), PS^{IR} has to forbid the reordering of RMWs with subsequent writes. Then, the mapping of certain RMW instructions to ARMv8 requires an extra "fake" control dependency from the read part of the RMW, so the hardware will not reorder RMWs with following plain writes (which arise from non-atomic writes in the source). We refer the reader to [Cho et al. 2021] for the exact mapping scheme and the (unnoticeable) performance impact of it. We note that for hardware that preserves load-store ordering for all stores, this additional fake dependency is not needed.

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

Definition 5.1. A function $o : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a program prog under* PS^{IR} if some execution of *prog* terminates with a memory M (*i.e.*, $\mathsf{init}(prog) \to^* \langle \mathcal{T}, P_G, M\rangle \wedge \mathsf{terminal}(\langle \mathcal{T}, P_G, M\rangle)$), and $o(X) = v$ where $\langle X@t, v, \_, \_\rangle \in M$ is the message to X with the greatest timestamp t.

Definition 5.2. A function $o : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a program prog under* Armv8S *(*PowerS*)* if o assigns to every location X the value of the co-maximal write to X in some execution graph of *prog* that is Armv8S-consistent (PowerS-consistent).[16]

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

Theorem 5.3. *For a* PS^{IR} *program prog, we denote by* $(\!|prog|\!)_{\mathsf{A}}$ *the* Armv8S *program obtained by mapping prog as described above. Then, given a program prog and an outcome o of* $(\!|prog|\!)_{\mathsf{A}}$ *under* Armv8S*, we have that either o is an outcome of prog under* PS^{IR} *or prog has undefined behavior under* PS^{IR} *(i.e., it has an execution reaching a machine state of the form* $\langle \bot, \_, \_\rangle$*).*

To prove this theorem, we utilized the operational model for Armv8 by Pulte et al. [2019], who also showed (in Coq) its equivalence to the declarative formulation of Armv8. We extended their operational model with strong relaxed accesses, reestablished the equivalence of the extended models, and proved (in Coq), using a simulation argument, that runs of this extended operational model reaching a certain outcome corresponds to runs of PS^{IR} that yield the same outcome.

We believe the standard mapping from PS^{IR} to PowerS (with relaxed stores compiled as strong stores) is sound as well. Formally establishing the soundness of this mapping, possibly using the IMM memory model [Podkopaev et al. 2019], is left as future work.

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

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

6 RELATED WORK

Our proposal for a concurrency semantics refines, simplifies, and combines existing ideas: catch-fire and preserving load-store ordering as in RC11 [Boehm and Adve 2008; Boehm and Demsky 2014; Lahav et al. 2017], an operational presentation of RC11 using the promising semantics without promises and certified promises as a speculation mechanism to allow load-store reordering [Kang et al. 2017], justifying compiler optimizations on non-atomics based on sequential reasoning [Cho et al. 2022] (see also [Zha et al. 2022]), and the operational Arm model as a bridge between the high-level semantics and the hardware model [Pulte et al. 2019]. We also rely on [Alglave et al. 2021, 2014] for the models of Power and Arm, the experimental data on observed behaviors, the Herd model checker, and the testing framework; and on [Ou and Demsky 2018] for the performance evaluation of different compilation schemes.

In particular, our $\mathrm{PS}^{\mathrm{IR}}$ model is inspired by the $\mathrm{PS}^{\mathrm{na}}$ model in [Cho et al. 2022]. The most significant difference between $\mathrm{PS}^{\mathrm{IR}}$ and $\mathrm{PS}^{\mathrm{na}}$ is that $\mathrm{PS}^{\mathrm{na}}$ allows also promises of *relaxed* writes, which makes $\mathrm{PS}^{\mathrm{na}}$ significantly more complex than $\mathrm{PS}^{\mathrm{IR}}$. First, in $\mathrm{PS}^{\mathrm{na}}$ a thread promises messages with specific timestamp, value, and view, while $\mathrm{PS}^{\mathrm{IR}}$ only maintains sets of locations that threads will write to in the future. (This is possible because promises of $\mathrm{PS}^{\mathrm{IR}}$ are needed only for race detection.) Second, unlike $\mathrm{PS}^{\mathrm{IR}}$, $\mathrm{PS}^{\mathrm{na}}$ allows a thread to *lower* or *split* their promises, which leads to substantial complications in proofs. Lastly, a non-atomic write in $\mathrm{PS}^{\mathrm{IR}}$ adds a single message to the memory, while in $\mathrm{PS}^{\mathrm{na}}$ multiple messages may be added by a single non-atomic write.

The idea to use an undefined value rather than catch-fire in order to validate load introduction comes from the LLVM (informal) model. To the best of our knowledge, the first attempt to apply this approach in a formal model was in [Chakraborty and Vafeiadis 2017], where previous read values can be revisited whenever relevant writes are executed. This requires a rather complicated event-structure-based model, which does not admit the DRF guarantee. Later improvements of this model [Chakraborty and Vafeiadis 2019; Moiseenko et al. 2020] admit DRF but *fail* to support load introduction. In turn, our $\mathrm{PS}^{\mathrm{IR}}$ model (following $\mathrm{PS}^{\mathrm{na}}$) applies this approach together with promises. We are not aware of any previous proof relating an in-order source model based on catch-fire to an IR model that is based on undefined values.

Other weak memory models were recently proposed (see, *e.g.*, [Jagadeesan et al. 2020; Jeffrey et al. 2022; Paviotti et al. 2020]), but they are all focused on generally allowing load-store reordering, while our models (both source and IR) allow it only for non-atomic accesses. Notably, supporting load introduction in these models is rather hard, and besides the promising models (*e.g.*, the recent version in [Cho et al. 2022]), we are not aware of any model that fully supports load-store reordering as well as load introduction. In particular, Jeffrey et al. [2022] observe a tension between the kind of temporal reasoning supported by their model and load-introduction.

In contrast, other work, *e.g.*, [Liu et al. 2021; Marino et al. 2011], propose SC as a concurrency semantics for programmers, and study its expected cost (which can be rather high). In turn, we believe that an in-order model enjoys the advantages of SC, while allowing for rather minimal performance overhead, provided that catch-fire for races on non-atomics is acceptable.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Chung-Kil Hur is the corresponding author. Minki Cho, Sung-Hwan Lee, and Chung-Kil Hur were supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03. Roy Margalit and Ori Lahav were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant numbers 1566/18 and 814/22).

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Coq development and supplementary material for this paper. https://sf.snu.ac.kr/promising-ir/

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

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2021. Safe-by-Default Concurrency for Modern Programming Languages. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 10 (Sept. 2021), 50 pages. https://doi.org/10.1145/3462206

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *ASPLOS*. ACM, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434285

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *PLDI*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1993498.1993522

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

Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

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

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

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

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *PLDI*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL, Article 68 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290381

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *PLDI*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *ISCA*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/3079856.3080206

Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *Proc. ACM Program. Lang.* 7, POPL, Article 53 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571246

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

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

Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *PLDI*. ACM, New York, NY, USA, 903–917. https://doi.org/10.1145/3519939.3523734

Received 2022-11-10; accepted 2023-03-31