# Modular Data-Race-Freedom Guarantees in the Promising Semantics

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

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

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

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

## Abstract

Local data-race-freedom guarantees, ensuring strong semantics for locations accessed by non-racy instructions, provide a fruitful methodology for modular reasoning in relaxed memory concurrency. We observe that standard compiler optimizations are in inherent conflict with such guarantees in general fully-relaxed memory models.

Nevertheless, for a certain strengthening of the promising model by Lee et al. that only excludes relaxed RMW-store reorderings, we establish multiple useful local data-race-freedom guarantees that enhance the programmability aspect of the model. We also demonstrate that the performance price of forbidding these reorderings is insignificant. To the best of our knowledge, these results are the first to identify a model that includes the standard concurrency constructs, supports the efficient mapping of relaxed reads and writes to plain hardware loads and stores, and yet validates several local data-race-freedom guarantees. To gain confidence, our results are fully mechanized in Coq.

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

***Keywords:*** Relaxed Memory Concurrency; Operational Semantics; Compiler Optimizations; Data Race Freedom

**ACM Reference Format:**
Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454082


*PLDI '21, June 20–25, 2021, Virtual, Canada*

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454082

## 1 Introduction

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

A fundamental programmability guarantee of this kind is DRF-SC [3]. It ensures that data-race free programs (avoiding races using locks or designated synchronization accesses) only exhibit SC behaviors. Crucially, data-race freedom (DRF), the premise of DRF-SC, is only required to hold *under SC*, allowing programmers to use this guarantee knowing nothing about the underlying complex model, but rather naively imagining standard interleaving semantics that follows the program order and employs a conventional memory.

Since SC is sometimes considered overly expensive to ensure efficient implementations (and as building blocks for establishing DRF-SC), more refined DRF guarantees have been studied in the last few years [12, 22]. Each of these guarantees is applicable on a different level of accesses—requiring more restrictive race-freedom conditions and resulting in stronger semantics guarantees. In particular, in models with release/acquire (RA) accesses, one aims to ensure RA semantics for programs that exhibit no races on accesses weaker than RA accesses. This guarantee, called DRF-RA [22], allows programmers to use (non-racy) weaker (and more efficient) accesses than RA accesses while knowing *only* the RA semantics. The latter, although weaker than SC, is much simpler than the full underlying model, and it admits several verification methods and tools, including model checkers and program logics [2, 21, 28, 29]. Similarly, on the level of

"relaxed" accesses, which are weaker than RA ones and intended to be compiled to plain machine loads and stores, a DRF guarantee with respect to an "in-order" RC11-like [30] semantics (with no load buffering behaviors) ensures in-order semantics when, under the in-order semantics, races on relaxed accesses are properly confined (see DRF-PF in [22] and DRF-RLX in [12]). Again, the benefit is significant: an in-order semantics like RC11 is significantly simpler than an "out-of-order" model in which reads can read from later writes, and like SC and RA, "in-order" models admit several verification methods [13, 14, 24, 40].

Nevertheless, the global nature of all DRF guarantees mentioned above makes them only applicable when the whole program admits the required race freedom premise. Software, however, is modularly developed, often without access to the full code. Moreover, benign races in carefully crafted concurrency libraries make the DRF guarantees futile for reasoning by clients that use these libraries, leaving them with no formal assurances applicable without a complete understanding of the underlying model (see Fig. 1 for an illustrative example).

This drawback of the DRF guarantees is addressed by more refined "local" guarantees that can be applied also on *parts* of a given program [15, 16]. In particular, a local DRF (LDRF, for short) guarantee allows one to conclude that accesses to *certain shared locations* have stronger semantics provided that when assuming stronger semantics *to these locations*, the program exhibits no races *on them*. The important practical consequence is that it is safe to assume that the client portion of the code is running under the stronger semantics when races are completely confined in the library code. Moreover, clients may rely on the synchronization guarantees provided by libraries to establish race freedom of their code while still understanding only the stronger semantics.

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

On the positive side, we observe that by disabling a certain problematic compiler optimization, an LDRF guarantee w.r.t. an "in-order" RC11-like semantics becomes achievable. Concretely, we identify that *RMW-store reordering*[1] is the source of the problem and show that by disabling only these reorderings one is able to validate a critical LDRF guarantee. In turn, for (naive formulations of) LDRF-RA/SC, disabling RMW-store reordering does not suffice, and we address the problem by slightly strengthening the (naive) race-freedom premise. The resulting guarantees are useful for modular reasoning (as demonstrated in §5), and we are not aware of any non-contrived example where this strengthened race-freedom condition makes a difference.

| | |
|---|---|
| $r_0 :=$ **pop_wait**$(S)$ | $r_1 :=$ **pop_wait**$(S)$ |
| **lock**() | **lock**() |
| process $r_0$ accessing $X$, $Y$ | process $r_1$ accessing $X$, $Y$ |
| **unlock**() | **unlock**() |

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

**Figure 1.** A simple example demonstrating the weakness of the global DRF-SC guarantee

To establish that forbidding RMW-store reordering and slightly strengthening the race-freedom premise suffice for establishing the LDRF guarantees, we demonstrate a particular model that satisfies the desiderata. Concretely, we prove that three LDRF guarantees are validated by PS2.1, a variant of the promising semantics, mentioned as a possible simplification of PS2 in [31, §4.4], that supports all standard (local and global) optimizations excluding RMW-store reordering.

In addition to the theoretical results, we empirically investigate the cost of forbidding RMW-store reorderings, and observe that it is negligible in practice. Current standard compilers are very conservative with reorderings of atomic accesses [35], and mainstream architectures, except Armv8, do not allow RMW-store reordering. Even in Armv8, it is relevant only for non-acquire *unconditional* RMWs (*i.e.*, **FADD** or **XCHG**, but not **CAS**), for which a "fake" branch instruction is needed to prevent the reordering. Since **FADD**s and **XCHG**s are not executed frequently and fake branching is relatively cheap [35], we expect the implementation cost to be negligible in practice. We have performed a sequence of experiments that validate this hypothesis (§6).

As for the LDRF guarantees, we formulate three guarantees, and prove them for PS2.1, where each of which provides the key lemma for establishing the next one: (1) LDRF-PF w.r.t. the *promise-free* (RC11-like) semantics allowing one to restrict "promises"—a special mechanism that accounts for

[1] RMW (read-modify-write) operations, such as compare-and-swap (**CAS**), fetch-and-add (**FADD**) and atomic exchange (**XCHG**), *atomically* perform a read followed by a write to the same location. Certain models—*e.g.*, C11 [42], the promising semantics [22], and Weakestmo [12]—allow the reordering of non-acquire RMWs with subsequent relaxed writes to a different location.

load-store reorderings in the promising semantics, which is undoubtedly the most complicated and hard to reason about component of the model; (2) LDRF-RA w.r.t. release/acquire semantics; and (3) LDRF-SC w.r.t. SC semantics.[2]

To conclude, our contributions are summarized as follows:

1. We show that the full set of compiler optimizations is inconsistent with local DRF guarantees (§2).
2. We establish the consistency of three local DRF guarantees (LDRF-PF, LDRF-RA, and LDRF-SC) and all standard optimizations excluding RMW-store reordering by proving that PS2.1 validates them all (§4).
3. We outline the applicability of local DRF for reasoning about client code, as well as library code (§5).
4. We empirically observe that the performance impact of disabling RMW-store reorderings is negligible (§6).

Our LDRF proofs in §4 are fully mechanized in Coq. The formalization is available in the accompanying artifact.

## 2 Local DRF in Weak Memory Models

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

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

By far, the most complicated aspect of a weak memory semantics is related to allowing load-store reordering of possibly racy independent relaxed accesses (a.k.a. load buffering behaviors). This is the source of the infamous "out-of-thin-air" problem [7], the reason why per-execution declarative models cannot work and more complicated event-structure-based models are needed instead [12, 37], and the only rationale behind "promises" in the promising semantics. To circumvent this complexity, one can use less efficient stronger models, such as RC11 [30], that conservatively forbid load-store reorderings altogether (by disallowing cycles in the union of the program order and the reads-from relation), and thus cannot map relaxed accesses to plain machine loads and stores in architectures like Arm.

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

**Definition 2.1.** A memory model $M$ is *in-order* if every behavior allowed by $M$ corresponds to a trace of memory accesses that respects the program order such that every read $r$ of value $v$ from location $X$ is justified by some write $w$ that writes $v$ to $X$ and appears in the trace *before* $r$.

This definition covers a wide variety of (not so weak) memory models including RC11, TSO [36], causal consistency [25, 26], the OCaml model in [15], and (of course) SC. It ensures a conceptually simple semantics and enables several verification approaches [13, 14, 24, 40].

A natural approach to allow "in-order" reasoning for a given program in a model with (fully) relaxed accesses is to use a DRF guarantee. When such guarantee is provided, one is able to assume in-order semantics for programs that under in-order semantics exhibit no races on accesses annotated as relaxed (so that the guarantee can be applied knowing nothing about the out-of-order part of the semantics). Moreover, as demonstrated in §1, for being applicable in a modular fashion (*e.g.*, in the presence of unrelated races induced by some library methods over which the client has no control), this guarantee has to be local.

To give a more precise statement of such a local DRF guarantee (but still keep the discussion general), consider an arbitrary model $M$ with relaxed reads/writes, intended to be compiled to plain machine accesses, and "strong relaxed" writes, intended to be compiled with barriers to forbid the hardware from reordering a load followed by a strong relaxed write.[3] Strong relaxed writes provide "in-order" semantics in the following sense: Every behavior allowed by $M$ corresponds to some trace of memory accesses that respects the program order such that: (i) every read $r$ of value $v$ from location $X$ is justified by some write $w$ that writes $v$ to $X$ and appears in the trace; and (ii) if $r$ is justified by $w$ that is strong relaxed, then $w$ should appear before $r$ in the trace. (Note that $M$ allows a read $r$ to be justified by a *relaxed* write that is executed after $r$.)

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

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

| | | |
|---|---|---|
| $a := L$ | $b := X$ // 1 ? | |
| $\mathbf{libfun_1}()$ | $\mathbf{libfun_2}()$ | (LDRF-LB) |
| $X^{\texttt{rlx}} := a$ | **if** $b = 1$ **then** $L := 1$ **else** $L^{\texttt{rlx}} := 1$ | |

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

[3] Strong relaxed accesses were introduced in [22] as a technical tool for establishing the correctness of mapping to hardware. They are also useful in the current discussion. Like release writes, they forbid reordering with preceding reads; but unlike release writes, they are not intended to synchronize with reads by other threads.

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

The compiler may optimize Thread 1 as shown below:

| (0) | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| $Y := 0$ | $Y := 0$<br>$c := L$ | $Y := 0$<br>$c := L$<br>**if** $c = 1$ **then** | $Y := 0$<br>$c := L$<br>**if** $c = 1$ **then** | $Y := 0$<br>$c := L$<br>**if** $c = 1$ **then** | $Y := 0$<br>$c := L$<br>**if** $c = 1$ **then** |
| $a := Y$<br>**if** $a \neq 0$ **then** | $a := Y$<br>**if** $a \neq 0$ **then** | $a := Y$ **else** $a := Y$<br>**if** $a \neq 0$ **then** | $a := Y$ **else** $a := 0$<br>**if** $a \neq 0$ **then** | $a := Y$<br>**if** $a \neq 0$ **then** | $a := Y$<br>**if** $a \neq 0$ **then** |
| $b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then** | $b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then** | $b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then** | $b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then** | $b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then** | $b := \mathbf{CAS}^{\mathsf{srlx}}(X, 0, 37)$ |
| $c := L$<br>**if** $c = 1$ **then**<br>$X^{\mathsf{srlx}} := 37$ | **if** $c = 1$ **then**<br>$X^{\mathsf{srlx}} := 37$ | **if** $c = 1$ **then**<br>$X^{\mathsf{srlx}} := 37$ | **if** $c = 1$ **then**<br>$X^{\mathsf{srlx}} := 37$ | *(eliminated)*<br>$X^{\mathsf{srlx}} := 37$ | |
| | | | | **else** $a := 0$<br>*(eliminated)* | **else** $a := 0$ |

(1) reorder the read $c := L$ to be second, after introducing the same read $c := L$ in the else-branches (when $b \neq 0$ or $a = 0$);
(2) insert a dummy if-then-else on $c = 1$ and distribute the rest of the code to both branches ("trace-preserving" transformation);
(3) forward the write $Y := 0$ to the read $a := Y$ in the else-branch, turning it into $a := 0$;
(4) distribute the branch on $a \neq 0$ to both prior branches on $c = 1$ and optimize them: eliminate repeated redundant testing of $c = 1$ in the then-branch, and remove dead code in the else-branch ("trace-preserving" transformation);
(5) merge $b := \mathbf{CAS}(X, 0, 42)$ and **if** $b = 0$ **then** $X^{\mathsf{srlx}} := 37$ into $b := \mathbf{CAS}^{\mathsf{srlx}}(X, 0, 37)$.

Now, the compiler may optimize Thread 2 as shown on the right:

(1) noticing that $X \neq 42$ is a global invariant (42 is never written to $X$), optimize away the redundant test "**if** $(d \neq 42)$ **then**";
(2) reorder the independent **CAS** on $X$ and write to $L$.

| (0) | (1) | (2) |
|---|---|---|
| $Y := 1$ | $Y := 1$ | $Y := 1$ |
| $d := \mathbf{CAS}(X, 0, 1)$ | $d := \mathbf{CAS}(X, 0, 1)$ | $L := 1$ |
| **if** $d \neq 42$ **then** | *(eliminated)* | |
| $L := 1$ | $L := 1$ | $d := \mathbf{CAS}(X, 0, 1)$ |

**Figure 2.** Program transformations on LDRF-PF-Fail (in the final transformed program, we may get $d = 37$ even under SC!)

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

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

| | | |
|---|---|---|
| $Y := 0$<br>$a := Y$<br>**if** $a \neq 0$ **then**<br>$b := \mathbf{CAS}(X, 0, 42)$<br>**if** $b = 0$ **then**<br>$c := L$<br>**if** $c = 1$ **then**<br>$X^{\mathsf{srlx}} := 37$ | $\parallel$ | $Y := 1$<br>$d := \mathbf{CAS}(X, 0, 1)$ *// 37 ?*<br>**if** $d \neq 42$ **then**<br>$L := 1$ |

(LDRF-PF-Fail)

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

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

Now we can demonstrate the inconsistency between the local DRF guarantee above and program optimizations. Since the premise of the guarantee is satisfied (for $\mathcal{L} = \{L\}$), if the guarantee holds, we may assume that $L := 1$ has strong relaxed semantics, and conclude, by the exact same reasoning, that the $d = 37$ outcome is disallowed (no execution executes both $c := L$ and $L := 1$). Nevertheless, Fig. 2 shows that starting from this program, a sequence of transformations, each of which is intended to be sound in standard weak memory models, may actually lead to the $d = 37$ outcome!

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

As a concrete example for a model $M$, consider the PS2 model [31], which satisfies the above assumptions and validates all transformations used in Fig. 2. (In PS2, strong relaxed writes correspond to relaxed writes that cannot be promised ahead of their execution.) It follows that PS2 fails to admit the above guarantee w.r.t. an in-order semantics.[6]

To locate the source of the problem, we observe that RMW-store reordering (applied in the second thread's code in Fig. 2) is the transformation that breaks a key property, which we call *promise monotonicity* (formally stated in §4.1), needed for our proof. Indeed, one of the main ideas in proving local DRF is to show that relaxed store hoisting (moving a relaxed write to be before other instructions) does not allow more behaviors unless the store was racy before the code motion. However, this property fails if reordering of a relaxed RMW followed by a relaxed write to a different location is allowed. For instance, in the program above, executing $d := \mathbf{CAS}(X, 0, 1)$ before $L := 1$ prevents the behavior executing both $L := 1$ and $c := L$, but executing them in the opposite order allows that behavior.

Accordingly, to accomplish our proof, we switched to PS2.1, a variant of the PS2 model outlined in [31, §4.4], which gives up RMW-store reordering for simplicity and better meta-theoretic properties such as the absence of deadlocking executions.[7] For PS2.1 we are able to prove LDRF-PF—a local DRF guarantee with respect to the promise-free fragment of the promising semantics (an in-order model), thus establishing the consistency of such a local DRF guarantee with all optimizations except for RMW-store reordering.

### 2.2 Local DRF w.r.t. RA and SC

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

A naive notion of LDRF-RA can be naturally derived from the global DRF-RA guarantee. The latter ensures that a program has only RA behaviors provided that under RA semantics it exhibits no races involving accesses annotated as (strong) relaxed [22]. To "localize" this guarantee with respect to a given set $\mathcal{L}$ of locations, we need to consider "$\mathcal{L}$-RA behaviors"—behaviors in which accesses to locations in $\mathcal{L}$ are treated as RA accesses (even when annotated with weaker modes), but other accesses are interpreted as annotated in the program. Then, a naive LDRF-RA guarantee would say that a program has only $\mathcal{L}$-RA behaviors provided that its $\mathcal{L}$-RA behaviors exhibit no races involving accesses to locations in $\mathcal{L}$ annotated with access modes weaker than release and acquire.

We show that in any sensible weak memory model this guarantee is actually inconsistent with standard program optimizations (here, RMWs are not involved at all). Specifically, the $\{L\}$-RA behaviors of the program on the right exhibits no races on the location $L$, but a sequence of standard optimizations may lead to a non $\{L\}$-RA behavior, which invalidates the naive LDRF-RA guarantee.

| | |
|---|---|
| $a := Y$ // 1 ? | |
| **if** $a = 1$ **then** | |
| $b := L$ | $c := X$ |
| $X := b$ | $L := 1$ |
| **else** | $Y := c$ |
| $X := 1$ | |

(Naive-LDRF-RA-Fail)

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

Second, with the same assumption (that the accesses to $L$ are RA), the above reasoning also shows that there are no races on $L$. In fact, the exact definition of a race does not matter here: we actually know that the first thread will not access $L$ at all. Then, the naive LDRF-RA for $\mathcal{L} = \{L\}$ implies that the program has only $\{L\}$-RA behaviors, which, as argued above, entails that the $a = 1$ outcome is disallowed. Nevertheless, in Fig. 3, we show that starting from the above program, a sequence of program transformations, each of which is intended to be sound in standard weak memory models, may actually lead to this outcome!

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

A possible way to do so is to weaken the semantics of "racy reads" from locations in $\mathcal{L}$ in the $\mathcal{L}$-RA semantics, and say that unlike standard acquire reads, these reads do not induce synchronization. However, this solution would require a precise definition of the semantics of racy reads, which goes beyond standard RA semantics.

In this paper, we follow an alternative approach that involves a certain over-approximation—we will say that a racy read simply invokes undefined behavior (UB). Since UB includes any possible behavior, the race-freedom condition based on racy reads invoking UB implies the one where racy

[6] The original promising model PS [22] does not admit global value-range analysis, which is needed in the sequence of transformations in Fig. 2. Nevertheless, in [1, Appendix A], we present a similar (yet more intricate) counterexample for PS.

[7] We have formally established the absence of deadlocks in Coq [1].

The compiler may optimize Thread 1 as shown below:

| (0) | (1) | (2) | (3) | (4) |
|---|---|---|---|---|
| | $b := L$ | $b := L$ | $b := L$ | $b := L$ |
| | | **if** $b = 1$ **then** | **if** $b = 1$ **then** | **if** $b = 1$ **then** |
| $a := Y$ | $a := Y$ | $a := Y$ | $a := Y$ | $X := 1$ |
| **if** $a = 1$ **then** | **if** $a = 1$ **then** | **if** $a = 1$ **then** | $X := 1$ | $a := Y$ |
| $b := L$ | | | | **else** ... |
| $X := b$ | $X := b$ | $X := b$ | | |
| **else** | **else** | **else** | | |
| $X := 1$ | $X := 1$ | $X := 1$ | | |
| | | **else** ... | **else** ... | |

(1) reorder the read $b := L$ to be first, after introducing the same read $b := L$ in the else-branch (when $a \neq 1$);
(2) insert a dummy if-then-else on $b = 1$ and distribute the rest of the code to both branches ("trace-preserving" transformation);
(3) in the then-branch on $b = 1$, substitute $b$ with 1 and merge both branches on $a = 1$ ("trace-preserving" transformation);
(4) reorder the independent read from $Y$ and write to $X$.

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

(1) reorder $c := X$ and $L := 1$.

| (0) | (1) |
|---|---|
| $c := X$ | $L := 1$ |
| $L := 1$ | $c := X$ |
| $Y := c$ | $Y := c$ |

**Figure 3.** Program transformations on Naive-LDRF-RA-Fail (after the transformations, we may get $a = 1$ even under SC!)

reads do not induce synchronization. In other words, we will say that a race occurs if some racy read is reachable ignoring what happens after the racy read is executed. With this definition, relying on the previously mentioned LDRF-PF as a key lemma, we proved LDRF-RA (and LDRF-SC) for PS2.1.[8] Importantly, unlike C11's "catch-fire" semantics, UB for races is not a part of the concurrency semantics (indeed, the promising semantics provides means to avoid "catch-fire"), but is only used for defining races when establishing the premise of LDRF-RA/SC. We note that a similar strengthening of the race-freedom premise in LDRF-PF does not solve the problem outlined in §2.1 (LDRF-PF-Fail is still a counterexample).

## 3 Preliminaries: The Promising Semantics

In this section we provide an introduction to the promising semantics. We include only the necessary parts for keeping our presentation self-contained, and refer the reader to [22, 31] for detailed explanations. Our focus is on the version described in [31, §4.4], which we refer to as PS2.1.[9]

We present the fragment of the model containing: *relaxed reads and writes* (`rlx`), *strong relaxed writes* (`srlx`), *release writes* (`rel`), and *acquire reads* (`acq`). Read-modify-writes (RMWs) carry two access modes—one for the read part and one for the write part. To simplify the presentation, we omit fences and release sequences. We also elide "system calls", used in [22, 31] to specify the observations of a given program. Instead, as we did when analyzing the examples above, we identify behaviors with final outcomes assigning values to certain registers. Nevertheless, our formal development in the artifact handles *all* features previously included in [22, 31] and uses system calls to define observable behaviors.

Figure 4 summarizes the different domains and (implicitly typed) metavariables. To define the **machine states**, besides a set Loc of locations and a set Val of values, we assume a set Time of *timestamps* which are rational numbers (totally and densely) ordered by $<$ with 0 being the minimum value. A *view*, $V \in \mathsf{Loc} \to \mathsf{Time}$, records a timestamp for each location. We represent half-open ranges of timestamps using *timestamp intervals* denoted by $(f, t]$ with $f < t$ or $f = t = 0$. A *machine state* is a pair $\langle \mathcal{T}, M \rangle$, where:

- $M$, called *memory*, is a finite set of *messages* and *reservations*. A message $m$ takes the form $\langle X @ (f, t], v, V \rangle$ where: $X \in \mathsf{Loc}$, $(f, t]$ is a timestamp interval ($t$ is called the *timestamp* of $m$), $v \in \mathsf{Val}$, and $V \in \mathsf{View}$ (called *message view*). In turn, a reservation $r = X @ (f, t]$ is defined like a message but without a value and a view. For a memory to be well-formed (as we implicitly assume henceforth), we require that messages/reservations with the same location have disjoint timestamp intervals; and that the view of each message is pointing to a timestamp of an existing message for every location. The initial memory consists of an initialization message $\langle X @ (0, 0], 0, \bot \rangle$ for every location $X$, where $\bot \triangleq \lambda X.\, 0$ denotes the bottom view.
- $\mathcal{T}$ is a mapping assigning a *thread state* $T = \langle \sigma, V, P \rangle$ to every thread $\pi \in \mathsf{Tid}$, where:
  - $\sigma$ records the (thread-local) *program state*. To keep the presentation abstract, rather than introducing a concrete syntax, we assume that the programming language is represented as a transition system, with local transitions labeled with the action that is performed. Each program state $\sigma$ consists of the program code, the current program counter and local register file. To run PS2.1 on a program *prog*, we initialize the program state of each thread to include its part of *prog* and the initial program counter and register file.
  - $V$, called the *thread view*, records the highest timestamp that the thread has observed for each location.
  - $P$, called the *thread promise set*, is a set of messages and reservations recording the thread's outstanding promised and reserved writes. Since every promise and reservation is also added to the memory, we will always have $P \subseteq M$.

Importantly, we require thread states to be *well-formed*, where for every location $X$, the current view of $\pi$ for $X$ is lower than the timestamp of all of $\pi$'s outstanding promised writes for $X$ (*i.e.*, $\langle X @ (\_, t], \_, \_ \rangle \in P \Rightarrow V(X) < t$). This condition, called *promise-consistency* in [31], is equivalent to saying that a thread should always be able to fulfill its promises by executing *some* sequence of operations (but not necessarily the sequence dictated by the program).

[8] PS2 does not satisfy our LDRF-RA/SC theorems (LDRF-PF-Fail is a counterexample for them as well).
[9] Most of the details, however, are identical for the original PS model and for the PS2 model (the only difference has to do with the notion of "capped memory" and reservations).

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

**Figure 4.** Domains and metavariables in PS2.1

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

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

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

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

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

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

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

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

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

**Figure 5.** Thread configuration steps in PS2.1

Figure 5 provides the **thread configuration steps**:

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

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

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

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

**RESERVE**. To support register promotion and a more efficient mapping of RMWs to Arm (see Example 3.3 below), PS2.1 (as well as PS2, but unlike PS) allows threads to reserve timestamp intervals for their own future writes. Unlike promises, reservations do not commit on the *value* that will be used to fill the reserved interval, and thus cannot be read by other threads. They are only used to "block" timestamp intervals in the memory. As in the promise step, a thread adds the reservation to both the memory and its promise set.

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

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

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

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

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

**Definition 3.1** (Consistency)**.** A thread configuration $\langle T, M\rangle$ is *consistent* if $\langle T, \widehat{M}\rangle \to^* \langle\langle \_, \_, \emptyset\rangle, \_\rangle$ where $\widehat{M}$, called *capped memory*, is the memory obtained from $M$ as follows:

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

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

Roughly speaking, consistency requires *certification*: the thread that took the step should be able to fulfill all its promises when executed in isolation. The certification starts from a *capped* version $\widehat{M}$ of the current memory $M$, where all timestamp intervals between existing messages and reservations are blocked by reservations and a "cap reservation" is attached to the message with the highest timestamp for each location. As demonstrated in Example 3.4 below, a consequence of this is that promises cannot be made across RMW operations. (This is where the PS2.1 and PS2 differ; see [31].)

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

**Remark 1.** In [31] the machine step consists of any sequence of thread steps. We observe (and proved in Coq) that by using reservations and cancellations, it is possible to obtain a "normal form" for machine steps: a (possibly empty) sequence of cancellations, followed by a *single* thread step, followed by a (possibly empty) sequence of reservations. This normal form simplifies modular reasoning, as we can assume a consistent state when control is passed between the library code and the client code.

Next, we present several instructive examples involving RMWs. We refer the reader to [22, 31] for more examples related to the basic views and promises mechanisms.

**Example 3.2.** Two competing RMWs can never read from the same message in memory, as the following annotated program demonstrates:

$$a := \textbf{FADD}(X, 1)\ \ /\!/ 0 \ \Big\|\ b := \textbf{FADD}(X, 1)\ \ /\!/ 0 \qquad \text{(Upd)}$$

Like **CAS**, we assume that **FADD** returns the value read before the update. Without loss of generality, suppose that $\pi_1$ executes first. As it performs an RMW operation, it must "attach" the message it adds to an existing message. Since the only existing message in this stage is the initial one $\langle X@(0, 0], 0, \bot\rangle$, $\pi_1$ will add $m = \langle X@(0, t], 1, \bot\rangle$ with some $t > 0$ to the memory. Then, the RMW of $\pi_2$ cannot also read from the initial message because this would require $\pi_2$'s message to be attached to the initial message, which would overlap with the $(0, t]$ interval of $m$.

**Example 3.3.** The following annotated program illustrates a drawback of the original PS that prevents register promotion and the intended mapping to Armv8 [18]:

$$\begin{array}{l} a := X\ \ /\!/ 1 \\ b := \textbf{FADD}^{\text{acqrel}}(Z, 1)\ \ /\!/ 0 \\ Y := 1 \end{array} \ \Big\|\ \begin{array}{l} c := Y\ \ /\!/ 1 \\ X := c \end{array} \qquad \text{(Arm-weak)}$$

The annotated behavior is allowed by Armv8 (for the compiled program), and can be also obtained if the thread-local location $Z$ is made a register. It is, however, disallowed by PS. PS2 and PS2.1 solve this problem using *reservations*. To observe $a = 1$, $\pi_1$ should be able to promise the write of 1 to $Y$ at the beginning of the execution. This is not possible without reservations because $\pi_1$ cannot update $Z$ during the certification against the capped memory. However, $\pi_1$ can *reserve* the interval $(0, 1]$ for the **FADD** before making the promise $Y = 1$. Then, it can certify the promise $Y = 1$ by using the reserved interval to perform the **FADD**. Intuitively speaking, while PS2.1 forbids the reordering of an RMW followed by a store, using reservations, it enables the reordering of the read part of the RMW before the read of $X$ and the write part of the RMW after the write of $Y$, which more faithfully captures Arm's load-linked/store-conditional implementation.

**Example 3.4.** The following annotated program shows a behavior forbidden by PS2.1 because of its stronger certification requirement w.r.t. PS and PS2.

$$\begin{array}{l} a := \textbf{FADD}(X, 1)\ \ /\!/ 1 \\ Y := 1 \end{array} \ \Big\|\ \begin{array}{l} b := Y \\ c := \textbf{FADD}(X, b) \end{array} \qquad \text{(RMW-W)}$$

For $\pi_1$ to read 1 via its **FADD**, it has to promise $Y = 1$. Unlike PS and PS2, this is not allowed in PS2.1 because $\pi_1$ cannot perform **FADD** to $X$ during the certification against the capped memory. Promising the **FADD** or reserving a space for it by $\pi_1$ is impossible as well. Once $\pi_1$ promises its **FADD**, it is committed to update $X$ from 0 to 1. If $\pi_1$ reserves a timestamp interval $(0, t]$ for its **FADD**, $\pi_2$ cannot update $X$ from 0 to 1 since the $X = 0$ message is blocked by $\pi_1$'s reservation, again forcing $\pi_1$ to update $X$ from 0 to 1.

## 4 Local DRF Guarantees

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

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

The supplementary material includes the statements of "time-wise" local DRF guarantees ([1, Appendix B]), which we do not discuss in the main text. Roughly speaking, these guarantees apply when no race occurs between two states

in the machine trace and they ensure the stronger semantics between these two states.

All results of this section (including time-wise LDRF) are fully mechanized in Coq (~35$K$ LoC altogether) [1].

### 4.1 Local DRF-PF

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

**Definition 4.1.** Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-PF-machine* is the strengthening of PS2.1 obtained by forbidding the application of the (PROMISE) rule for locations in $\mathcal{L}$. We denote by $[\![prog]\!]^{\mathcal{L}}_{\mathrm{PF}}$ the set of all behaviors of a program *prog* allowed by the $\mathcal{L}$-PF-machine.

Next, we define what a racy execution in the $\mathcal{L}$-PF-machine is. Roughly, an execution is $\mathcal{L}$-racy if it includes some thread $\pi_1$ taking a machine step writing a message $m$ to a location in $\mathcal{L}$ by a relaxed write, immediately followed by another thread $\pi_2$ taking a sequence of machine steps that ends with reading the message $m$.

**Definition 4.2.** An execution in the $\mathcal{L}$-PF-machine is *$\mathcal{L}$-racy* if it includes a sequence of machine steps of the form:

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

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

Then, LDRF-PF is formulated as follows.

**Theorem 4.3** (LDRF-PF)**.** *If there is no $\mathcal{L}$-racy execution of prog in the $\mathcal{L}$-PF-machine, then $[\![prog]\!]_{PS2.1} = [\![prog]\!]^{\mathcal{L}}_{PF}$.*

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

Revisiting LDRF-PF-Fail, the argument outlined in §2 shows that no execution of LDRF-PF-Fail in the $\{L\}$-PF-machine is $L$-racy. Then, from Thm. 4.3, it follows that the $d = 37$ outcome is disallowed for that program under PS2.1.

**Example 4.4.** As an instructive example of an application of LDRF-PF, we show that no execution of the following program in the $\{X, Y\}$-PF-machine is $\{X, Y\}$-racy, and so $[\![prog]\!]_{\mathrm{PS2.1}} = [\![prog]\!]^{\{X,Y\}}_{\mathrm{PF}}$.

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

Clearly, there is no race on $Y$ since the program has no relaxed writes to $Y$ (syntactically). Now, assuming no promises on $X$ and $Y$, the write to $Z$ in $\pi_2$ can neither be promised nor executed before $\pi_1$ executes the write to $Y$. Similarly, the read from $X$ in $\pi_3$ cannot be executed before $\pi_2$ promises or executes the write to $Z$. Therefore, the write to $Y$ in $\pi_1$ should be first executed in order for $\pi_3$ to execute the read from $X$, and thus there is no $X$-racy execution in the $\{X, Y\}$-PF-machine.

***Proof sketch of LDRF-PF.*** We highlight the main ideas in the proof of Thm. 4.3, which is the most challenging among our results. For its proof, we introduce an intermediate semantics, called *$\mathcal{L}$-PRF-machine*, and define the notion of race in this machine (PRF stands for promise-read-free).

**Definition 4.5.** Given a set $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-PRF-machine* is the strengthening of PS2.1 obtained by forbidding steps reading from promises to locations in $\mathcal{L}$. We denote by $[\![prog]\!]^{\mathcal{L}}_{\mathrm{PRF}}$ the set of all behaviors of a program *prog* allowed by the $\mathcal{L}$-PRF-machine. *$\mathcal{L}$-racy* executions in the $\mathcal{L}$-PRF-machine are defined exactly as in Def. 4.2.

Then, we prove the following three lemmas for every program *prog*, from which Thm. 4.3 directly follows:

(I) $[\![prog]\!]^{\mathcal{L}}_{\mathrm{PRF}} \subseteq [\![prog]\!]^{\mathcal{L}}_{\mathrm{PF}}$.
(II) If there is no $\mathcal{L}$-racy execution of *prog* in the $\mathcal{L}$-PRF-machine, then $[\![prog]\!]_{\mathrm{PS2.1}} \subseteq [\![prog]\!]^{\mathcal{L}}_{\mathrm{PRF}}$.
(III) If there is an $\mathcal{L}$-racy execution of *prog* in the $\mathcal{L}$-PRF-machine, then there is one in the $\mathcal{L}$-PF-machine.

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

To prove (II), we use the following "reshuffling" mechanism: when thread $\pi_1$ can take a sequence *seq* of thread steps reading a promise $m$ of another thread $\pi_2$ to a location $L \in \mathcal{L}$, we first execute $\pi_2$ following its certification until it fulfills the promise $m$ and then execute $\pi_1$ following *seq* until it reads $m$. What makes this possible is Lemma 4.6 below. Using "reshuffling", (II) is established as follows. Roughly speaking, ignoring the consistency requirement, for the first time a thread can read from a promise on a location in $\mathcal{L}$, we apply the above construction to get an $\mathcal{L}$-racy execution without reading any promise on $\mathcal{L}$ (*i.e.*, a $\mathcal{L}$-racy execution in the $\mathcal{L}$-PRF-machine), which contradicts the premise of (II). (To meet the consistency requirement, the proof requires repeated applications of the reshuffling.)

**Lemma 4.6** (Promise Monotonicity)**.** *Let $\langle \mathcal{T}, M \rangle$ be a (consistent) machine state with a promise $m$ written by thread $\pi_1$. Suppose that $\langle \mathcal{T}(\pi_2), M \rangle \rightarrow^{*} \xrightarrow{l} \langle T_2, \_ \rangle$ for some thread $\pi_2 \neq \pi_1$, label $l$, and thread state $T_2$. Then, there exist $l_m \in \{\mathtt{W}(\mathtt{rlx}, m), \mathtt{RMW}(\_, \mathtt{rlx}, \_, m)\}$ and memory $M_1$ such that:*

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

**Remark 3.** Promise consistency does not hold for PS and PS2 since RMW-store reordering breaks it. For *global* DRF-PF, a weaker property, which does hold for PS and PS2, suffices. Specifically, the above reshuffling may break during the execution of $\pi_1$ following the sequence *seq* (before it reads $m$) only if $\pi_1$ performs a racy RMW. Global DRF-PF follows from the race on the RMW, but not LDRF-PF since the location of the RMW may not be in $\mathcal{L}$.

### 4.2 Local DRF-RA

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

**Definition 4.7.** Given a set $\mathcal{L} \subseteq \mathrm{Loc}$, the *$\mathcal{L}$-RA-machine* is the strengthening of the $\mathcal{L}$-PF-machine obtained by interpreting all accesses to $\mathcal{L}$ as if they have release/acquire access modes (in (READ-HELPER) and (WRITE-HELPER)). We denote by $[\![prog]\!]^{\mathcal{L}}_{\mathrm{RA}}$ the set of all behaviors of a program *prog* that are allowed by the $\mathcal{L}$-RA-machine.

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

**Definition 4.8.** The *$\mathcal{L}$-RA-race-detecting-machine* is the machine obtained from the *$\mathcal{L}$-RA-machine* by adding following thread configuration step:

$$\frac{\begin{array}{c} L \in \mathcal{L} \qquad \lambda \in \{\mathtt{R}(o_{\mathtt{R}}, L, \_), \mathtt{RMW}(o_{\mathtt{R}}, \_, L, \_, \_)\} \qquad \sigma \xrightarrow{\lambda} \_ \\ V(L) < t \qquad m = \langle L@(\_, t], \_, \_\rangle \in M \\ o_{\mathtt{R}} = \mathtt{rlx} \vee m \text{ was written by a non-release write}^{10} \end{array}}{\langle\langle \sigma, V, \_\rangle, M\rangle \xrightarrow{\mathtt{race}} \langle\langle \bot, V, \emptyset\rangle, M\rangle}$$

**Remark 4.** A similar view-based definition of a race can be also used in LDRF-PF. However, such definition would unnecessarily deem too many programs as racy, resulting in a weaker guarantee. For example, with a view-based definition of a race in LDRF-PF, we would not be able to show the absence of $\{X, Y\}$-PF-racy executions for the program in Example 4.4 (since there is no synchronization from the write to $X$ in $\pi_1$ to the read from $X$ in $\pi_3$).

LDRF-RA is formulated as follows.

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

**Theorem 4.9** (LDRF-RA)**.** *If the* race *transition is never enabled in runs of the $\mathcal{L}$-RA-race-detecting-machine on prog, then $[\![prog]\!]_{PS2.1} = [\![prog]\!]^{\mathcal{L}}_{RA}$.*

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

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

Then, the global DRF-RA guarantee follows from the local one. The "naive" LDRF-RA discussed in §2.2 (which cannot not hold together with all optimizations allowed in PS2.1) formally means to use the above step for race detection in the $\mathcal{L}$-RA-race-detecting-machine.

**Example 4.10.** The following example is a variant of the common "load-buffering" test. We show that, using LDRF-RA, this program never exhibits the $a = 1$ outcome.

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

Assuming RA semantics for $X$, both the writes to $X$ and to $Y$ cannot be promised, and clearly $a = 1$ is not allowed. Now, we show that the race transition is never enabled in executions of this program in the $\{X\}$-RA-race-detecting-machine. Indeed, since the write to $X$ in $\pi_2$ can only be executed after the write to $Y$ in $\pi_1$ is executed (which cannot be promised because it is a release write), there cannot be any message to $X$ except for the initial message before the read from $X$ in $\pi_1$ is executed.

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

**Example 4.11.** We apply LDRF-RA on a location with a write-write race. In the following program, the first two threads access $X$ and $Y$ and raise flags $Z$ and $W$. The third thread waits on both flags and then accesses $X$ and $Y$.

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

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

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

$$\begin{array}{l} r_1 := X \\ \vdots \\ r_n := X \\ \mathbf{fence}^{\mathsf{acq}} \end{array} \quad \leftrightsquigarrow \quad \begin{array}{l} r_1 := X^{\mathsf{acq}} \\ \vdots \\ r_n := X^{\mathsf{acq}} \\ \mathbf{fence}^{\mathsf{acq}} \end{array} \qquad \begin{array}{l} \mathbf{fence}^{\mathsf{rel}} \\ X_0 := r_0 \\ X_1 := r_1 \\ \vdots \\ X_n := r_n \end{array} \quad \leftrightsquigarrow \quad \begin{array}{l} \mathbf{fence}^{\mathsf{rel}} \\ X_0^{\mathsf{rel}} := r_0 \\ X_1^{\mathsf{srlx}} := r_1 \\ \vdots \\ X_n^{\mathsf{srlx}} := r_n \end{array}$$

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

### 4.3 Local DRF-SC

The final LDRF guarantee, LDRF-SC, provides the strongest semantics for non-racy accesses, but also requires much more for accesses to be considered non-racy. We note that, unlike C/C++11 [6, 8, 30], the promising semantics does not provide sequentially consistent accesses (it only has SC fences). Thus, a *global* DRF-SC can only pointlessly ensure SC semantics for programs that have no races whatsoever (with no mechanism to actually avoid races). Nevertheless, *local* DRF-SC is still meaningful as it only requires to avoid races on certain locations.

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

**Definition 4.12.** In the context of a machine state, we call a message *maximal* if there does not exist a message with the same location and higher timestamp. For $\mathcal{L} \subseteq \mathsf{Loc}$, the *$\mathcal{L}$-SC-machine* is the strengthening of the *$\mathcal{L}$-RA-machine* obtained by requiring that for every $L \in \mathcal{L}$:

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

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

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

**Definition 4.13.** The *$\mathcal{L}$-SC-race-detecting-machine* is the machine obtained from the $\mathcal{L}$-SC-machine by adding following thread step:

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

Then, LDRF-SC is formulated as follows.

**Theorem 4.14** (LDRF-SC)**.** *If the* `race` *transition is never enabled in runs of the $\mathcal{L}$-SC-race-detecting-machine on prog, then $[\![prog]\!]_{PS2.1} = [\![prog]\!]^{\mathcal{L}}_{SC}$.*

**Example 4.15.** Consider the message passing program:

$$\begin{array}{l} D := 42 \\ F^{\mathsf{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := F^{\mathsf{acq}} \\ \mathbf{if}\ a = 1\ \mathbf{then} \\ \quad a := D \quad /\!/ 42 \end{array} \qquad \text{(MP)}$$

In all its executions in the $\{D\}$-SC-race-detecting-machine, the view of $\pi_2$ for $D$ after reading 1 from $F$ points to the message $D = 42$ written by $\pi_1$. Therefore, the `race` transition is never enabled for this program in the $\{D\}$-SC-race-detecting-machine. Then, LDRF-SC with $\mathcal{L} = \{D\}$ ensures SC semantics on the location $D$.

## 5 Applying LDRF for Modular Reasoning

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

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

### 5.1 Reasoning About Client Code

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

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

| | | |
|---|---|---|
| **push**(5) | $r_0 :=$ **pop_wait**() | $r_1 :=$ **pop_wait**() |
| **push**(7) | **lock**() | **lock**() |
| | $s_0 := S$ | $s_1 := S$ |
| | $S := s_0 + r_0$ | $S := s_1 + r_1$ |
| | **unlock**() | **unlock**() |

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

**lock**() $\triangleq$
  **do** $a :=$ **CAS**$^{\texttt{acqrel}}(L, 0, 1)$
  **while** $(a \neq 0)$

**unlock**() $\triangleq$
  $L^{\texttt{rel}} := 0$

Further, suppose that the collection library guarantees that **push** and **pop_wait** (syntactically) never access $S$ and $L$, and that when the same number of push and pop are invoked, the values returned by **pop_wait** are in some one-to-one correspondence with the values pushed by **push**.[11]

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

(i) By the locality property, we can safely ignore the impact on $S$ and $L$ by **push** and **pop_wait**;
(ii) Since $L$ is only accessed by RA accesses, we know that there are not any races on $L$;
(iii) The lock specification (specifically the RA synchronization from **unlock**() to **lock**()) ensures that a thread accessing $S$ always has the maximal view on $S$, so the accesses to $S$ are not racy as well.

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

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

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

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

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

**enque**$(d)$ $\triangleq$
  **lock**()
  $t := T^{(\texttt{acq})}$
  **if not** $t < 32$ **then**
    **unlock**(); **return** `full`
  $D[t]^{(\texttt{rel})} := d$
  $T^{\texttt{rel}} := t + 1$
  **unlock**(); **return** 0

**try_deque**() $\triangleq$
  $t := T^{\texttt{acq}}$
  $b := B^{\texttt{acq}}$
  **if not** $b < t$ **then**
    **return** `empty`
  $d := D[b]^{(\texttt{acq})}$
  $b' :=$ **CAS**$^{\texttt{acqrel}}(B, b, b + 1)$
  **return** $(b = b' \ ? \ d \ : \ \texttt{fail})$

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

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

(i) By the locality property, we can safely ignore the impact on $U, D, T, B, L$ by **log**;
(ii) Since $D, T, B, L$ are only accessed by RA accesses, there are not any races on them;
(iii) The queue specification ensures a synchronization from an **enque** writing to $D[k]$ to a **try_deque** reading from $D[k]$ for any $k$ via the accesses to $T$, since the **enque** writes $k+1$ to $T$, the **try_deque** reads some $k' > k$ from $T$, and all the writes to $T$ are synchronized via **lock**() and **unlock**();
(iv) It also ensures that each successful **try_deque** returns a unique index due to the atomicity of **CAS** in **try_deque**;
(v) From these, it follows that each **use**$(U, i)$ accesses disjoint locations, and since the synchronization on $T$ ensures no races on $U$ between write **write**$(U, i)$ and **use**$(U, i)$, we avoid races on $U$ as well.

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

### 5.2 Reasoning About Library Code

Next, we describe how LDRF-RA can be used to reason about the implementation of the queue library above. We consider an implementation of the above specification that simply lowers the accesses in parentheses, (`acq`) and (`rel`), to be

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

`rlx` accesses. (This optimization may be significant if the size of each cell in $D$ is large.)

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

(i) By the locality property, we can safely ignore the impact on $D, T, B, L$ by the context;
(ii) Since $B, L$ are only accessed by RA accesses, there are not any races on them;
(iii) For $T$, the only possible race is between the `rlx` read and the `rel` write in **enque**, which, however, reside in the same locked region thereby avoiding race;
(iv) For $D$, the reasoning in §5.1(iii) for the client program applies, thereby avoiding races on $D[k]$ for any $k$.

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

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

## 6 Mapping PS2.1 to Hardware

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

PS2.1 supports the intended compilation schemes to mainstream architectures [11], but for Armv8, it requires an additional (fake) control dependency from the read part ("load-linked") of each fetch-and-add and exchange instruction with relaxed read mode. The compilation schemes for these instructions along with the more optimal schemes are given in [1, Appendix C].[12] Lee et al. [31, Section 6.5] established (in Coq) the correctness of these schemes from PS2 to hardware models using the Intermediate Memory Model, IMM [39]. We observe here that their proof works as is for PS2.1.[13] We note that compared to PS, PS2.1 still supports more efficient mapping of RMW operations, which for PS require an "ld fence" barrier that is more expensive than a control dependency (see Example 3.3).

[12]Our compilation schemes employ standard LL/SC-style RMW implementations. We leave to future work the evaluation of an implementation that uses Armv8.1's LSE (Large System Extension) for RMWs [4].
[13]The proof does not handle atomic exchange instructions, which are not supported in IMM.

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

**Figure 6.** Performance overhead for each benchmark (The "Average" column denotes the arithmetic mean and the 95% confidence interval.)

We believe that forbidding RMW-store reorderings only mildly affects performance since: (*i*) standard compilers do not aggressively reorder RMWs with atomic writes [35]; (*ii*) with the exception of Armv8, mainstream hardware (x86-TSO, POWER, Armv7, and RISC-V) do not reorder such accesses; and (*iii*) the performance overhead in Armv8 for forbidding this optimization is negligible.

We demonstrate (*iii*) by evaluating 19 highly concurrent data structures with extensive use of fetch-and-add and exchange operations selected from the CDS C++ library [23], as well as four artificial "worst case litmus tests" that repeatedly perform fetch-and-add and exchange operations. Specifically, the four tests consist of following: FADD/XCHG tests where each thread repeatedly performs **FADD**/**XCHG** to a single location; and FADD-RW/XCHG-RW tests where each thread repeatedly performs **FADD**/**XCHG** to a single location followed by load/store to another location ($n/2$ threads load and $n/2$ threads store).

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

Figure 6 summarizes the performance overhead for each benchmark (see [1, Appendix C] for more detailed results). We conclude that there is no statistical evidence for a noticeable performance cost induced by the suboptimal RMW compilation of PS2.1.

**Remark 7.** During the evaluation, we identified a bug in LLVM's compilation of exchange instructions to Armv8. When the value read by the exchange instruction is never

used, LLVM 10.0.0 compiles C++11 relaxed exchange instructions into Arm's plain *store* instructions. However, since an acquire fence may induce synchronization when it follows an exchange instruction, but *not* when it follows a store, this optimization is unsound: it may introduce behaviors in the compiled Armv8 assembly that are not allowed for the C++ source program. A concrete example of the miscompilation is provided in [1, Appendix D].

## 7 Conclusion and Related Work

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

We believe that it may be useful to study existing and novel models through the lens of our results also beyond the context of the promising semantics. A "just right" programming language shared-memory concurrency model that is not too strong to allow efficient implementation and not too weak to program with has been the subject of extensive research in recent years (*e.g.*, [7–9, 12, 15, 19, 20, 27, 30, 32–34, 37, 38, 41–44]). While implementability is nowadays relatively well-defined, the criteria for the programmability aspect are much less evident. Since, as we argue in this paper, local DRF guarantees facilitate modular software development, these guarantees provide better programmability desideratum than the standard global DRF properties. Our impossibility result shows an inherent trade-off that has to be considered when designing a memory model, while the positive result assigns the blame on RMW-store reorderings.

Several papers have previously studied local DRF guarantees. Dolan et al. [15] introduced local DRF-SC, and established such guarantees for a model with two types of access. Their guarantees account for two aspects of "locality": (*i*) in terms of "space", which is similar to our location-wise LDRF above and (*ii*) in terms of "time", which we cover in [1, Appendix B]. However, their memory model is much stronger than the one studied here. In particular, it is an "in-order" model (see Def. 2.1), as even their weak accesses completely forbid load-store reorderings (including RMW-store reorderings), and cannot be compiled to plain machine accesses on architectures like Arm. In addition, their strong accesses are stronger than C11's SC accesses, so that strong stores have to be mapped to atomic exchanges even on x86.

Dongol et al. [16] established local DRF-SC guarantees for a model more general than the one of [15] with multiple access modes. In their model, threads synchronize via software transactions with RA semantics. While release/acquire RMWs can be implemented as transactions, the problematic relaxed RMWs are not expressible in the model of [16], so that our impossibility result does not apply.

Recently, Jagadeesan et al. [19] presented a denotational concurrency model and sketched (without full proofs) a time-wise local DRF-SC guarantee for a fragment of this model that does not include fences and RMWs. (They presented several variants, and their reported LDRF result is for a version that does not support irrelevant load introduction.) Their model is multi-copy-atomic, and thus, unlike PS2.1, it cannot be efficiently compiled to POWER or Armv7.

We note that the strengthening of accesses in the "SC machines" (used for detecting races for the LDRF-SC premise) in prior work [15, 16, 19] does not make them synchronizing (inducing "happens-before" w.r.t. other locations). Thus, like ours, previous local DRF-SC theorems are weaker than the naive formulation discussed in §2.2.

Finally, while modular reasoning about libraries in weak memory semantics has been studied in multiple papers, *e.g.*, [5, 10, 17, 40], to the best of our knowledge, the observation that location-wise local DRF guarantees are essential for such reasoning is lacking in prior work. We leave to future work the development of LDRF-based formal tools, which will allow one to formalize (and possibly mechanize) reasoning as we did in §5. In particular, our LDRF-PF paves the way for the application of program logics for an "in-order" semantics (*e.g.*, the logic in [13] that essentially targets the PF model), which is significantly simpler than any semantics allowing load-buffering behaviors. We also note that while the applications in §5 are for RA-centric specifications, our local DRF results are generally applicable for weaker library specifications as well. Nevertheless, it is currently unclear how to formally specify and verify libraries in memory models that allow load buffering behaviors (as was studied in [40] for 'in-order' models). We leave this question as well for future work.

## Acknowledgments

We thank Anton Podkopaev for his help with the Coq proofs of mapping from PS2.1 to IMM. We thank our shepherds, Nathan Chong and James Riely, and the anonymous PLDI reviewers for their helpful 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-IT1502-53. Ori Lahav was supported by the Israel Science Foundation (grant number 1566/18) and by the Alon Young Faculty Fellowship. This research was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811).

## References


[1] 2021. Coq development and supplementary material for this paper. https://sf.snu.ac.kr/promising-ldrf

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

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

[4] Arm. 2020. Arm A64 Instruction Set Architecture Armv8 (DDI0596 2020-12). https://developer.arm.com/documentation/ddi0596/2020-12

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

[6] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *POPL*. ACM, New York, NY, USA, 634–648. https://doi.org/10.1145/2837614.2837637

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

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

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

[10] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *ESOP*. Springer, Berlin, Heidelberg, 87–107.

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

[12] 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

[13] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

[14] 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

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

[16] Brijesh Dongol, Radha Jagadeesan, and James Riely. 2019. Modular Transactions: Bounding Mixed Races in Space and Time. In *PPoPP*. ACM, New York, NY, USA, 82–93. https://doi.org/10.1145/3293883.3295708

[17] Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On Abstraction and Compositionality for Weak-Memory Linearisability. In *VMCAI*. Springer International Publishing, Cham, 183–204.

[18] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *POPL*. ACM, New York, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

[19] 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

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

[21] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[22] 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

[23] Max Khiszinsky. 2020. *CDS C++ Library*. https://github.com/khizmax/libcds

[24] 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

[25] Ori Lahav. 2019. Verification under Causally Consistent Shared Memory. *ACM SIGLOG News* 6, 2 (April 2019), 43–56. https://doi.org/10.1145/3326938.3326942

[26] Ori Lahav and Udi Boker. 2020. Decidable Verification under a Causally Consistent Shared Memory. In *PLDI*. ACM, New York, NY, USA, 211–226. https://doi.org/10.1145/3385412.3385966

[27] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

[28] 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

[29] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP*. Springer, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

[30] 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

[31] 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

[32] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[33] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2016. DRFx: An Understandable, High Performance, and Flexible Memory Model for Concurrent Languages. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 16 (Sept. 2016), 40 pages. https://doi.org/10.1145/2925988

[34] 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

[35] 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

[36] 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

[37] 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

[38] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *POPL*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

[39] 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

[40] 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

[41] 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

[42] 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

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

[44] Yang Zhang and Xinyu Feng. 2016. An Operational Happens-before Memory Model. *Front. Comput. Sci.* 10, 1 (Feb. 2016), 54–81. https://doi.org/10.1007/s11704-015-4492-4