Promising 2.0: Global Optimizations in Relaxed Memory Concurrency

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

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

Anton Podkopaev
National Research University Higher School of Economics & MPI-SWS
Russia & Germany
podkopaev@mpi-sws.org

Soham Chakraborty
IIT Delhi
India
soham@cse.iitd.ac.in

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

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

Viktor Vafeiadis
MPI-SWS
Germany
viktor@mpi-sws.org

Abstract

For more than fifteen years, researchers have tried to support global optimizations in a usable semantics for a concurrent programming language, yet this task has been proven to be very difficult because of (1) the infamous "out of thin air" problem, and (2) the subtle interaction between global and thread-local optimizations.

In this paper, we present a solution to this problem by redesigning a key component of the *promising semantics* (PS) of Kang et al. Our updated PS 2.0 model supports all the results known about the original PS model (*i.e.*, thread-local optimizations, hardware mappings, DRF theorems), but additionally enables transformations based on global value-range analysis as well as register promotion (*i.e.*, making accesses to a shared location local if the location is accessed by only one thread). PS 2.0 also resolves a problem with the compilation of relaxed RMWs to ARMv8, which required an unintended extra fence.

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

Keywords: Relaxed Memory Concurrency; Operational Semantics; Compiler Optimizations

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3386010

ACM Reference Format:
Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386010

1 Introduction

A major challenge in programming language semantics has been to define a weak memory model for a concurrent programming language supporting efficient compilation to the mainstream hardware platforms (*i.e.*, x86, POWER, ARMv7, ARMv8, RISC-V) including all applicable compiler optimizations and yet avoiding semantics quirks, such as "out of thin air" reads [16], that prevent formal reasoning about programs and break DRF guarantees (the latter provide simpler semantics to data-race-free programs). In particular, such a semantics must allow the following annotated outcome (assuming all variables are initialized to zero and all accesses are relaxed).

$$\begin{array}{l|l} a := x \;\; /\!/ 1 & b := y \;\; /\!/ 1 \\ y := 1 & x := b \end{array} \qquad \text{(LB)}$$

This outcome is observable after a compiler transformation that reorders the (independent) accesses of thread 1, while on ARM [20] it is even observable without the transformation.

While there are multiple partial solutions to this challenge [7, 8, 12, 16, 18], none of them properly supports global compiler optimizations, namely program transformations whose validity depends on some global analysis. Examples of such transformations are (a) removal of null pointer checks based on global null-pointer analysis; (b) removal of array bounds checks based on global size analysis; and (c) *register promotion*, *i.e.*, converting accesses to a shared variable that happens to be used by only one thread to local accesses. The latter is very important in languages like Java that have only

atomic accesses, but is also useful for C/C++. For instance, in single-threaded programs, it allows the removal of locks, as well as the promotion to register accesses of inlined function calls of concurrent data-structures.

The desire to support global optimizations in concurrent programming languages goes at least as back as 15 years ago with the Java memory model (JMM) [16]. In fact, the very first JMM "causality test case" is centered around value-range analysis. Assuming all variables are initialized to 0, JMM allows the annotated outcome of the following example:

$$\begin{array}{l} a := x \;\; /\!/ 1 \\ \textbf{if } a \geq 0 \textbf{ then} \\ \quad y := 1 \end{array} \;\Big\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(JMM1)}$$

> *"Decision: Allowed, since interthread compiler analysis could determine that x and y are always non-negative, allowing simplification of $a \geq 0$ to true, and allowing write $y := 1$ to be moved early."* [10]

Supporting global optimizations, however, is rather challenging because of their interaction with local transformations. Global optimizations generally depend on invariants deduced by some global analysis but these invariants need not hold in the source program; they might hold after some local transformations have been applied. In the following example, (only) after the local elimination of the overwritten $x := 42$ assignment, the condition $a < 10$ becomes a global invariant, and so can be simplified to true as in JMM1.

$$\begin{array}{l} a := x \;\; /\!/ 1 \\ \textbf{if } a < 10 \textbf{ then} \\ \quad y := 1 \end{array} \;\Big\|\; \begin{array}{l} x := 42 \\ b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(LB-G)}$$

In more complex cases, a global optimization may enable a local transformation, which may further enable another global optimization, which may enable another local optimization, and so on. As a result, supporting both global and local transformations is very difficult, and none of the solutions so far has managed to fully support global analysis along with all the expected thread-local transformations.

In this paper, we present the first memory model that solves this challenge: (i) it allows the aforementioned global optimizations (value-range analysis and register promotion); (ii) it validates the thread-local compiler optimizations that are validated by the C/C++11 model [13] (*e.g.*, roach-motel reorderings [21]); (iii) it can be efficiently mapped to the mainstream hardware platforms (x86, POWER, ARMv7, ARMv8, RISC-V); and (iv) it supports reasoning principles in the form of DRF guarantees, allowing programmers to resort to simpler well-behaved models when data races are appropriately restricted. In developing our model we mainly use (i)–(iii) to conclude that some behavior should be allowed; while (iv) tells us which behaviors must be forbidden.

As a starting point, we take the *promising semantics* (PS) of Kang et al. [12], a concurrency semantics that satisfies *almost* all our desiderata. It supports almost all C/C++11 features, all expected thread-local compiler optimizations, and several DRF theorems. In addition, Podkopaev et al. [19] established the correctness of a mapping from PS to hardware.[1] The main drawback of PS is that it does not support global optimizations.

PS is an operational semantics which represents shared memory as a set of messages (*i.e.*, writes). To support out-of-order execution, PS employs a non-standard step, allowing a thread to *promise* to perform a write in the future, which enables other threads to read from it before the write is actually executed.

The technical challenge resides in identifying the exact conditions on such promise steps so that basic guarantees (like DRF and no "thin-air values") are maintained.

In PS, these conditions are completely thread-local: the thread performing the promise must be able to run in isolation from *all extensions* of the current state and fulfill all its outstanding promises. While thread-locality is useful, quantifying over all extensions of the current state prevents optimizations based on global analysis because some extensions may well not satisfy the invariant produced by the analysis.

Checking for promise fulfillment only from the current state without extension enables global analysis, but breaks the DRF guarantee (see §4). Our solution is therefore to check promise fulfillment for a carefully crafted extension of the current state, which we call *capped memory*. Because capped memory does not contain any new values, it is consistent with optimizations based on global value analysis. However, it still does not allow optimizations like register promotion.

To support register promotion, we introduce *reservations*, which allow a thread to secure an exclusive right to perform an atomic read-modify-write instruction reading from a certain message without fixing the value that it will write (because, for example, that might not have yet been resolved). In addition, reservations resolve a problem with the compilation of PS to ARMv8, whose intended mapping of RMWs was unsound and required an extra fence [19].[2]

With these two new concepts, we are able to retain the thread-local nature of PS and yet fully support global optimizations and the intended mapping of RMWs along with all the results available for PS. Our redesigned PS 2.0 model is the first weak memory model that achieves these results. To establish confidence in our model, we have formalized our key results in the **Coq proof assistant**.

Outline. In the following, we first review the PS definition (§2), and why it does not support global optimizations

[1] Albeit, the mapping of RMWs to ARMv8 contains one more barrier ("ld fence") than intended because the intended mapping is unsound.

[2] Our current mechanized proof requires a fake control dependency from relaxed fetch-and-add instructions, which is currently not added by standard compilers. We believe that the compilation from our model without this dependency is sound as well, and leave the formal proof to a future work (see also §6.5).

(§3). We then present our PS 2.0 model both informally in an incremental fashion (§4) and formally all together (§5). In §6, we establish the correctness of mappings from PS 2.0 to hardware, and show that PS 2.0 supports all the local transformations and reasoning principles known to be allowed by PS, as well as register promotion, and the introduction of 'assert' statements for invariants derived by global analysis. The mechanization of our main results in Coq, the full model definitions, and written proofs of additional claims are available in [1].

2 Introduction to the Promising Semantics

In this section, we introduce the promising semantics (PS) of Kang et al. [12]. For simplicity, we present only a fragment of PS containing only three kinds of memory accesses: *relaxed* (the default mode), *release writes* (`rel`), and *acquire reads* (`acq`). Read-modify-write (RMW) instructions, such as compare-and-swap (**CAS**) and fetch-and-add (**FADD**), carry two access modes—one for the exclusive read and one for the write. We put aside other access modes, fences, and release sequences, as they are orthogonal to the contribution of this paper. We refer the reader to [12] for the full PS model.

Domains. We assume non-empty sets Loc of locations and Val of values. We also assume a set Time of *timestamps*, which is totally and densely ordered by $<$ with 0 as its minimum. (In our examples, we take non-negative rational numbers as timestamps with their usual ordering.) A *view*, $V \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$, records the largest known timestamp for each memory location. A timestamp *interval* is a pair of timestamps $(f, t]$ with $f < t$ or $f = t = 0$. It represents the range of timestamps from (but not including) f up to and including t.

Memory. In PS, the memory is a set of *messages* representing all previously executed writes. A message m is of the form $\langle x : v@(f, t], R\rangle$, where $x \in \mathsf{Loc}$ is the location, $v \in \mathsf{Val}$ is the stored value, $(f, t]$ is a timestamp interval, and $R \in \mathsf{View}$ is the message view. The latter is used to model release-acquire synchronization and will be explained shortly. Initially, the memory consists of an initialization message for every location carrying the value 0, the interval $(0, 0]$, and the bottom view $\bot \triangleq \lambda x.\ 0$. We require that any two messages with the same location in memory have disjoint timestamp intervals. The timestamp (also called the *"to"-timestamp*) of a massage $\langle x : v@(f, t], R\rangle$ is the upper bound t of the message's timestamp interval. The lower bound f, called the *"from"-timestamp*, is needed to handle atomic updates (a.k.a. RMW operations) as explained below.

Machine State. PS is an operational model where threads execute in an interleaved fashion. The *machine state* is a pair $\Sigma = \langle \mathcal{TS}, M\rangle$, where $\mathcal{TS}$ assigns a *thread state* TS to every thread and M is a (global) *memory*. A thread state is a triple $TS = \langle \sigma, V, P\rangle$ where σ is the local store recording the values of its local variables, $V \in \mathsf{View}$ is the *thread view*, and P is a set of messages representing the thread's outstanding promises.

Relaxed Reads and Writes. Thread views are instrumental in providing correct semantics to memory accesses. The thread view, V, records the "knowledge" of each thread, *i.e.*, the timestamp of the most recent message that it has observed for each location. It is used to forbid a thread to read from a (stale) message m if the thread is aware of a "newer" message, *i.e.*, when $V(x)$ is greater than the message's timestamp. Similarly, when a thread adds messages of location x to the memory, it has to pick a timestamp t for the added message that is greater than its view of x ($V(x) < t$):

READ. A thread can read from memory M by simply observing a message $\langle x : v@(f, t], \_\rangle \in M$ provided that $V(x) \le t$, and updating its view for x to t.

WRITE. A thread adds a new message $m = \langle x : v@(f, t], \bot\rangle$ to the memory where the timestamp t is greater than the thread's view of x ($V(x) < t$) and there is no other message with the same location and overlapping timestamp interval in the memory. Relaxed writes set the message view to $\bot$, which maps each location to timestamp 0.

The following example illustrates how timestamps of messages and views interact. Note that we assume that both threads start with the initial thread view that maps x and y to 0, and that every location is initialized to 0: the initial memory only contains messages $\langle x : 0@(0, 0], \bot\rangle$ and $\langle y : 0@(0, 0], \bot\rangle$.[3]

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ a := y \ \ /\!/ 0 & b := x \ \ /\!/ 0 \end{array} \qquad \text{(SB)}$$

Here, both threads are allowed to read from the initialization messages, 0. When thread 1 performs the write to x, it will add a message $\langle x : 0@(f, t], \bot\rangle$ by choosing some $t > f \ge 0$. During this write, thread 1 should increase its view of x to t, while maintaining $V(y)$ to be 0 as it was. Hence, thread 1 is still allowed to read 0 from y in the subsequent execution. As thread 2 can be executed in the same way, both threads are allowed to read 0.

Relaxed Atomic Updates. Atomic updates (a.k.a. RMW operations) are essentially a pair of accesses to the same location—a read followed by a write—with an additional atomicity guarantee: the read reads from a message that immediately precedes the one added by the write. PS employs timestamp intervals (rather than single timestamps) to enforce atomicity.

UPDATE. When a thread performs an RMW, it first reads a message $\langle x : v@(f, t], \bot\rangle$, and then writes the updated message with "from"-timestamp equal to t, *i.e.*, a message of the form $\langle x : v'@(t, t'], \bot\rangle$. This results in consecutive messages

[3] In all our code examples, we assume that all memory accesses are relaxed (`rlx` memory order) unless annotated otherwise.

$(f, t], (t, t']$, forbidding other writes to be later placed between the two messages (recall that messages with the same location must have disjoint timestamp intervals).

This constraint, in particular, means that two competing RMWs cannot read from the same message, as the following "parallel increment" example demonstrates.[4]

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

Without loss of generality, suppose that thread 1 executed 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 $\langle x : 0@(0, 0], \bot\rangle$, thread 1 will first add a message $m = \langle x : 1@(0, t], \bot\rangle$ with some $t > 0$ to the memory. Then, the RMW of thread 2 cannot also read from the initial message because its interval would overlap with the $(0, t]$ interval of m. Therefore, the annotated behavior is forbidden. More abstractly speaking, the timestamps intervals of PS express a dense total order on messages to the same location together with immediate adjacency constraints on this order, which are required for handling RMW operations.

Release and Acquire Accesses. To provide the appropriate semantics to release and acquire accesses, PS uses the message views. Indeed, a release write should transfer the current knowledge of the thread to other threads that read the message by an acquire read. Thus, (i) a release write operation puts the current thread view in the message view of the added message; and (ii) an acquire read operation incorporates the view of the message being read in the thread view (by taking the pointwise maximum).

READ is defined the same as before, except that when the thread performs an *acquire* read, it increases its view to contain not only the ("to") timestamp of the message read but also the view of that message.

WRITE is defined as before, except that *release* writes record the thread view in the message being added, whereas relaxed writes record the $\bot$ view.

As a result, the acquiring thread is confined in its future reads at least as the releasing thread was confined when it "released" the message being "acquired". As a simple example, consider the following:

$$\begin{array}{l} x := 1 \\ y^{\mathsf{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := y^{\mathsf{acq}} \;\; /\!/ 1 \\ \textbf{if } a = 1 \textbf{ then} \\ \quad b := x \;\; /\!/ 0 \end{array} \qquad \text{(MP)}$$

Here, if thread 2 reads 1 from y, which is written by thread 1, both threads are synchronized through release and acquire. Thus, thread 2 obtains the knowledge of thread 1, namely its view for x is increased to include the timestamp of $x := 1$ of thread 1. Therefore, after reading 1 from y, thread 2 is not allowed to read the initial value 0 from x.

[4]Here and henceforth, we assume that RMW instructions such as **FADD** and **CAS** return the value that was read during the read-modify-write operation (before the update).

Release/acquire RMW operations also transfer thread views via message views as release writes and acquire reads do.

Promises. The main novelty of PS lies in its way to enable the reordering of a read followed by a write (of different locations), needed to explain the outcome of the LB program in §1. Thus, besides step-by-step program execution, PS allows threads to non-deterministically *promise* their future writes. This is done by simply adding a message (whose interval does not overlap with that of any existing message to the same location) to the memory. Later, the execution of write instructions may also *fulfill* an existing promise (rather than add a message to the memory). Thread promises are kept in the thread state, and removed when the promise is fulfilled. Naturally, at the end of the execution all promises must be fulfilled.

PROMISE. At any point, a thread can add a message to both its set of promises and the memory.

FULFILL. A thread can fulfill its promise by executing a (non-release) write instruction, by removing a message from the thread's set of promises. PS does not allow release writes to be promised, *i.e.*, a promise cannot be fulfilled through a release write instruction.

In the LB program above, thread 1 may promise $y := 1$ at first. This allows thread 2 to read 1 from y and write it back to x. Then, thread 1 can read 1 from x, which was written by thread 2, and fulfill its promise.

Certification. To ensure that promises do not make the semantics overly weak, each sequence of steps by a thread (before "yielding control to the scheduler") has to be *certified*: the thread that took the steps should be able to fulfill all its promises when executed in isolation. Indeed, revisiting the LB program above, note that at the point of promising $y := 1$ (in the very beginning of the run), thread 1 can run and perform $y := 1$ without any "help" of other threads.

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

$$\begin{array}{l} a := x \;\; /\!/ 1 \\ y := a \end{array} \;\Big\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(OOTA)}$$

As every thread simply copies the value it reads, both threads are not supposed to read any other value than 0 from the memory. However, the annotated behavior, often called out-of-thin-air, is allowed in C11 [3]. In PS, if a thread could promise without certification, this behavior would be allowed by the same execution as the one for LB. However, with the certification requirement, thread 1 cannot promise $y := 1$, as, when running in isolation, thread 1 will only write $y := 0$.

PS requires a certification to exist for *every future memory* (*i.e.*, any memory that extends the current memory). In §3, we explain the reason for this condition and its consequences.

Machine Step. A *thread configuration* $\langle TS, M\rangle$ can take one of **READ**, **WRITE**, **UPDATE**, **PROMISE**, and **FULFILL** steps, denoted by $\langle TS, M\rangle \to \langle TS', M'\rangle$. In addition, a thread configuration is called *consistent* if for every future memory M_{future} of M, there exist TS' and M' such that (where $TS.\texttt{prm}$ denotes the set of outstanding promises in thread state TS):

$$\langle TS, M_{\text{future}}\rangle \to^* \langle TS', M'\rangle \ \wedge \ TS'.\texttt{prm} = \emptyset$$

In turn, the machine step is defined as follows:

$$\frac{\langle \mathcal{TS}(i), M\rangle \to^+ \langle TS', M'\rangle \quad \langle TS', M'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

We note that the machine step is completely *thread-local*: it is only determined by the local state of the executing thread and the global memory, independently of the other threads' states. Thread-locality is a key design principle of PS. It is what makes PS conceptually well-behaved, and, technically speaking, it allows one to prove the validity of various local program transformations, which are performed by compilers and/or hardware, using standard thread-local simulation arguments.

To show a concrete example, we list the execution steps of PS leading to the annotated behavior of the LB program (items prefixed with "C" represent certification steps):

(1) Thread 1 promises $\langle y : 1@(1, 2], \bot\rangle$.
(C1) Starting from an arbitrary extension of the current memory, thread 1 reads $\langle x : 0@(0, 0], \bot\rangle$, the initial message of x.
(C2) Thread 1 fulfills its promise $\langle y : 1@(1, 2], \bot\rangle$.
(2) Thread 2 reads $\langle y : 1@(1, 2], \bot\rangle$.
(3) Thread 2 writes $\langle x : 1@(1, 2], \bot\rangle$.
(4) Thread 1 reads $\langle x : 1@(1, 2], \bot\rangle$.
(C1) Starting from an arbitrary extension of the current memory, Thread 1 fulfills its promise $\langle y : 1@(1, 2], \bot\rangle$.
(5) Thread 1 fulfills its promise $\langle y : 1@(1, 2], \bot\rangle$.

DRF-RA Guarantee. We end this introductory section by informally describing DRF-RA, one of the main programming guarantees provided by PS. Generally speaking, DRF guarantees ensure that race-free programs have strong (*i.e.*, more restrictive) semantics. To be more applicable and allow their use without even knowing the weaker semantics, race freedom is checked assuming the strong semantics.

In particular, DRF-RA is focused on release/acquire semantics (RA), and states that: if under RA semantics some program P has no data race involving relaxed accesses (*i.e.*, all races are on `rel`/`acq` accesses), then all behaviors that PS allows for P are also allowed for P by the RA semantics. Here, (i) by RA semantics we mean the model obtained from PS by treating *all* reads as `acq` reads, all writes as `rel` writes, and all RMWs as `acqrel`; and (ii) as PS is an operational model, data-races are naturally defined as states in which two different threads can access the same location and at least one of these accesses is writing.

For example, by analyzing the MP example under RA semantics, one can easily observe that the only race is on the `rel`/`acq` accesses to y. (Importantly, such analysis safely ignores promises, since these are not allowed under RA.) Then, DRF-RA implies that MP has only RA behaviors. In contrast, in the LB example, non-RA behaviors are possible, and, indeed, under RA semantics, there are races on relaxed accesses (to both x and y).

In the sequel, DRF-RA provides us with the main guideline for making sure that our semantics is not overly weak (that is, we exclude any semantics that breaks DRF-RA). DRF-RA also serves as a main step towards "DRF-Lock", which states that properly locked programs have only sequentially consistent semantics.[5]

3 Problem Overview

As we will shortly demonstrate, the main challenge in PS is to come up with an appropriate thread-local condition for certifying the promises made by a thread. Maintaining thread-locality is instrumental in proving correctness of many compiler transformations, but is difficult to achieve given that promises of different threads may interact.

As we briefly mentioned above, PS requires a certification to exist for any memory that extends the current memory. We start by explaining why certifying promises only from the current memory (without quantifying over all future memories) is not good enough. Kang et al. [12] observed that such model may deadlock: the promising thread may fail to fulfill its promise since the memory was changed since the promise was made. In this work, we observe that a model that requires certifying promises only from the current memory has much more severe consequences. It actually *breaks the DRF-RA guarantee* as illustrated below:

$a := \textbf{FADD}^{\texttt{acqrel}}(x, 1)$ // 0	$b := \textbf{FADD}^{\texttt{acqrel}}(x, 1)$ // 0
if $a = 0$ **then**	**if** $b = 0$ **then**
$y := 1$	$c := y$ // 1
	if $c = 1$ **then**
	$x := 0$

(CDRF)

Under RA semantics only one thread can enter the if-branch, and the only race is between the two **FADD**s. Hence, to maintain DRF-RA, we need to disallow the annotated behavior where both threads read 0 from x. To prevent this behavior, we need to disallow thread 1 to promise $y := 1$ in the beginning of the run. Indeed, by reading such a promise, thread 2 can write $x := 0$, and then, thread 1 can perform its update to x and fulfill its outstanding promise. However, if we completely ignore the possible interference by other

[5] The more standard DRF-SC, guaranteeing sequentially consistent semantics when all races (assuming SC semantics) are on SC accesses, is not applicable here since PS lacks SC accesses. The extension of PS with SC accesses is left to future work.

threads, thread 1 may promise $y := 1$, as it can be certified in a local run of thread 1 that starts from the initial memory and reads the initial message of x.

Abstractly, what went wrong is that two threads compete on the same resource (*i.e.*, to perform an RMW reading from the initialization message); one of them makes a promise assuming it will get the resource first but the other thread wins the competition in the actual run. This not only causes deadlock (which is semantically inconsequential), but also breaks DRF-RA.

To address this, PS followed a simple approach: it required that threads certify their promises starting from *any* extension of the current memory. One such particular extension is the memory that will arise when the required resource is acquired by some other thread. Hence, this condition does not allow threads to promise writes assuming they will win a competition on some resource.

Revisiting CDRF, PS's certification condition blocks the promise of $y := 1$. For example, when certifying against M_{future} that, in addition to the initialization messages, consists of a message $m = \langle x : 42@(0, \_], \_\rangle$, thread 1 is forced to read from m when performing its **FADD**, and cannot fulfill its promise. Since M_{future} is a possible future memory of the initial memory, thread 1 cannot promise $y := 1$.

PS's future memory quantification maintains the thread-locality principle and suffices for establishing DRF-RA. However, next, we demonstrate that this very conservative over-approximation of possible interference is too strong to support global optimizations, and it is also the source of unsoundness of the intended compilation scheme to ARMv8.

Value-Range Analysis. PS does not support global optimizations based on value-range analysis. To see this, consider a variant of the LB-G program above that does not have the redundant store to x in thread 2 and has a **CAS** instruction in thread 1.

$$\begin{array}{l} a := \mathbf{CAS}(x, 0, 1) \;\; /\!/ 1 \\ \mathbf{if}\ a < 10\ \mathbf{then} \\ \quad y := 1 \end{array} \;\Bigg\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(GA)}$$

In PS, the annotated behavior is disallowed. Indeed, to obtain this behavior, thread 1 has to promise $y := 1$. This promise, however, cannot be certified for every future memory M_{future}. For example, if, in addition to the initialization messages, the future memory M_{future} consists of a single message of the form $\langle x : 57@(0, \_], \_\rangle$, then the **CAS** instruction can only read 57, and the write $y := 1$ is not executed. However, by observing the global invariant $x < 10 \wedge y < 10$, a global compiler analysis may transform this program to the following:

$$\begin{array}{l} a := \mathbf{CAS}(x, 0, 1) \;\; /\!/ 1 \\ y := 1 \end{array} \;\Bigg\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array}$$

Now, the annotated behavior is allowed (the promise $y := 1$ is not blocked anymore), rendering the optimization unsound. This is particularly unsatisfying because PS ensures that $x < 10$ is globally valid in this program (via its "invariant logic" [12, §5.5]), but does not allow an optimizing compiler to make use of this fact.

Register Promotion. A similar problem arises for a different kind of global optimization, namely *register promotion*:

$$\begin{array}{l} a := x \;\; /\!/ 1 \\ c := \mathbf{FADD}(z, a) \;\; /\!/ 0 \\ y := 1 + c \end{array} \;\Bigg\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(RP)}$$

PS disallows the annotated behavior. Again, thread 1 cannot promise $y := 1$, since an arbitrary future memory may not allow it to read $z = 0$ when performing the RMW. (Note also the RMW writing $z := 1$ cannot be promised before $y := 1$ since it requires to read $x := 1$ first.) Nevertheless, a global compiler analysis may notice that z is a local variable in the source program, and perform register promotion, replacing $c := \mathbf{FADD}(z, a)$ with $c := 0$ (since this **FADD** always returns 0). Now, PS *allows* the annotated behavior (nothing blocks the promise $y := 1$), rendering register promotion unsound.

Unsound Compilation Scheme to ARMv8. A different problem in PS, found while formally establishing the correctness of compilation to ARMv8 [19], is that the intended mapping of RMWs to ARMv8 is broken. In fact, this problem stems from the exact same reason as the two problems above.

While PS disallows the annotated behavior of the RP program above, when following the intended mapping to ARMv8 [6], ARMv8 allows the annotated behavior for the target program.[6] Roughly speaking, although the instructions cannot be reordered at the source level, they can be reordered at the micro-architecture level. **FADD** is effectively turned into *two* special instructions, a load exclusive followed by a store exclusive. Since there is no dependency between the load of x and the exclusive load of z, the two loads could be executed out of order. Similarly, the two stores could be executed out of order, and so the store to y could effectively be executed before the load of x, which in turn leads to the annotated behavior.

What went wrong? These three problems all arise because PS's certification requirement against every memory extension is overly conservative in approximating the interference by other threads. The challenge lies in relaxing this condition in a way that will ensure the soundness of global optimizations while maintaining *thread-locality*.

As CDRF shows, simply relaxing the certification requirement by requiring certification only against the current memory is not an option. Another naive remedy would be to restrict the certification to extensions of the current memory that can actually arise in the given program. This approach, however, is bound to fail:

[6] Here the fact that no other thread accesses z is immaterial. ARMv8 allows this behavior also when, say, a third thread executes $z := 5$.

- First, due to the intricate interaction with local optimizations, a precise approximation of other threads effect on memory is too strong—we may have a preceding local optimization that reduces the behaviors of the other threads. For instance, consider the following program:

$$\begin{array}{l|l} a := \mathbf{CAS}(x,0,1) \;\; /\!/ 1 & x := 42 \\ \textbf{if } a < 10 \textbf{ then} & b := y \;\; /\!/ 1 \\ \quad y := 1 & x := b \end{array} \qquad \text{(GA+E)}$$

  Here, $x := 42$ occurs in a possible future memory, but a compiler may soundly eliminate this write.
- Second, this approach is not thread-local, and, since other threads may promise as well, it immediately leads to troublesome cyclic reasoning: whether thread 1 may promise a write depends on behavior of thread 2 that may include promise steps that again depend on behavior of thread 1.

4 Solution Overview

In this section, we present the key ideas behind our modified PS model, which we call PS 2.0. Section 4.1 describes the notion of *capped memory*, which enables value-range analysis, while §4.2 discusses *reservations*, an additional mechanism needed to support register promotion and recover the correctness of the mapping to ARMv8. Section 4.3 discusses our modeling of undefined behavior (which we use to formally specify value range analysis). Finally, §4.4 describes certain trade-offs in our model.

4.1 Capped Memory

We note that PS's certification against every memory extension is quantifying over two aspects of possible interference: message *values* and message *views*.

We observe that quantifying only over message views suffices for DRF-RA. By carefully analyzing CDRF, we can see that for DRF-RA, one has to make sure that during the certification of promises, no *acquire-release RMW* reads from a message that already exists in the memory. Indeed, (i) due to interference by other threads, such RMW may not have the opportunity to read from that message in the actual run; and (ii) such racy RMWs may exist (the DRF-RA assumption does not prevent them). Together, (i) and (ii) invalidate the DRF-RA guarantee (as happens in CDRF). We observe here that this is the *only* role of the future memory quantification that is required for ensuring DRF-RA.

The conservative future memory quantification of PS indeed disallows such problematic RMWs during certification. In fact, even certification against memory extensions that do not introduce new values in the future memory suffices for DRF-RA. For example, in CDRF, when certifying against M_{future} that, in addition to the initialization messages, has a message form $m = \langle x : 0@(0, \_], R\rangle$ with $R(y) \geq t$, thread 1 is forced to read m when performing its **FADD**. Since it is an acquire **FADD**, it will increase the thread view of y to $R(y)$, which will not allow it to fulfill its promise. More generally, when a thread promises a message of the form $\langle x : v@(f, t], V\rangle$ in the current memory M, there is always a possible memory extension M_{future} of M that forces (non-promised) RMWs of location y performed during certification (which read from a message in M_{future}) to read from a specific message $m^{y}_{\text{future}} \in M_{\text{future}}$ whose view of x is greater than or equal to t. When such RMWs are *acquire* RMWs, this will force the thread to increase its view of x to at least t, which, in turn, does not allow the thread to fulfill its promise.

Remark 1. Completely disallowing release-acquire RMWs during certification is too strong. We should allow them to read from local writes added during certification, since no other thread can prevent them from doing so.

We further observe that value-range analysis concerns message *values*, but it is *insensitive* to message *views*. As we saw for the GA program above, the conservative future memory quantification of PS is doing too much: it forbids any promise that depends on the value read by an RMW, which invalidates value-range analysis. However, we note that there is no problem in disallowing the following variant of GA that uses an acquire CAS instead of a relaxed one:

$$\begin{array}{l|l} a := \mathbf{CAS}^{\text{acq}}(x,0,1) \;\; /\!/ 1 & b := y \;\; /\!/ 1 \\ \textbf{if } a < 10 \textbf{ then} & x := b \\ \quad y := 1 & \end{array} \qquad \text{(GAacq)}$$

Although value analysis may deduce that $a < 10$ is always true, it cannot justify the reordering of $a := \mathbf{CAS}^{\text{acq}}(x, 0, 1)$ and $y := 1$, since acquire accesses in general cannot be reordered with subsequent accesses. In other words, an analysis that is based solely of values does not give any information about the views of read messages, so that any optimization based on such analysis cannot enable reordering of acquire RMWs.

Based on these observations, it seems natural to replace the conservative future memory quantification of PS with a requirement to certify against all extensions of the current memory M that employ values that already exist in M (for each location). While this approach makes value-range analysis sound and maintains DRF-RA, it is still too strong for the combination of local and global optimizations. Indeed, consider the following variant of the GA+E program above.

$$\begin{array}{l|l|l} x := 42 & f := flag & b := y \;\; /\!/ 1 \\ x := 0 & \textbf{if } f = 1 \textbf{ then} & x := b \\ flag^{\text{rel}} := 1 & \quad a := \mathbf{CAS}^{\text{acq}}(x,0,1) \;\; /\!/ 1 & \\ & \quad \textbf{if } a < 10 \textbf{ then} & \\ & \quad\quad y := 1 & \end{array} \qquad \text{(GA+E')}$$

In order for thread 2 to promise $y := 1$, the write to $flag$ has to be executed first. (Note that a release write cannot be promised.) Therefore, the value 42 for x exists in memory when the promise $y := 1$ is made, but, to support both the elimination of overwritten values and global value analysis, $x := 42$ should not be considered as a possible extension of

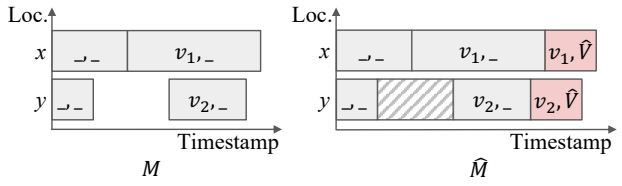

Figure 1. An example of the capped memory

the current memory. We observe that it is enough, however, to consider memory extensions whose additional messages *only* use values of maximal messages (which were not yet overwritten) to each location.

Now, instead of quantifying over a restricted set of memory extensions, we identify the most restrictive such extension, which we called the *"capped memory"*. This leads to a conceptually simpler certification condition, where certification is needed only against one particular memory, which is uniquely determined by the current memory. The capped memory $\widehat{M}$ of a memory M is obtained by:

- Filling all "gaps" between existing messages so that non-promised RMWs can only read from the maximal message of the relevant location. In other words, for every two messages $m_1 = \langle x : \_@(\_, t], \_\rangle$ and $m_2 = \langle x : \_@(f, \_], \_\rangle$ with $t < f$ and no message in between, we block the space between t and f. (The exact mechanism to achieve this, "reservations", is discussed in §4.2.)
- For every location x, attaching a "cap message" $\widehat{m}_x$ with a *globally maximal view* to the latest message to x in M:

$$\widehat{m}_x = \langle x : \widehat{v}_x@(\widehat{t}_x, \widehat{t}_x + 1], \widehat{V}_M\rangle$$

where $\widehat{t}_x$ and $\widehat{v}_x$ are the "to"-timestamp and the value of the message to x in M with the maximal "to"-timestamp, and $\widehat{V}_M$ is given by:

$$\widehat{V}_M = \lambda y.\ \max\{t \mid \langle y : \_@(\_, t], \_\rangle \in M\}.$$

Fig. 1 depicts an example of the capped memory construction. The shaded area in $\widehat{M}$ represents the blocked space.

Starting from $\widehat{M}$, any (non-promised) RMWs reading from a message in $\widehat{M}$ are forced to read from the $\widehat{m}_x$ messages (since the timestamp interval $[0, \widehat{t}_x]$ is completely occupied). Because these messages carry maximal views, acquire RMWs reading from them cannot be executed during certification, as it will increase the thread view to $\widehat{V}_M$, which, in turn, will prevent the thread from fulfilling its outstanding promises.

In turn, the new machine step is then simplified as follows:

$$\frac{\langle \mathcal{TS}(i), M\rangle \to^+ \langle TS', M'\rangle \quad \exists TS''.\ \langle TS', \widehat{M'}\rangle \to^* \langle TS'', \_\rangle \wedge TS''.\mathtt{prm} = \emptyset}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

Since the capped memory is clearly one possible future memory, the semantics we obtain is clearly weaker than PS. It is (i) weak enough to allow the annotated behaviors of GA and RP above: certification against the capped memory will not lead to $a \geq 10$ in GA and to $c \neq 0$ in RP; and, on the other hand, (ii) strong enough to forbid the annotated behavior of CDRF above: certification against the capped memory will not allow the $y := 1$ promise. In particular, by using the maximal messages for constructing capped memory, thread 2 of GA+E' can promise $y := 1$ and certify it while the message $x := 42$ (which is overwritten by $x := 0$) is in the memory.

Remark 2. The original PS quantification over all future memories could equivalently quantify over all memories defined just like the capped memory, except for using arbitrary values for the cap messages. Capped memory is more than that: it sets the value of each cap messages to that of the corresponding maximal message.

4.2 Reservations

While capped memory suffices for justifying the weak outcomes of the examples seen so far, it is still too strong to support register promotion and to validate the intended mapping to ARMv8. Consider the following variant of RP that uses an *acquire* RMW in thread 1.

$$\begin{array}{l} a := x \ \ /\!/ 1 \\ c := \mathbf{FADD}^{\mathsf{acq}}(z, a) \ \ /\!/ 0 \\ y := 1 \end{array} \ \Big\|\ \begin{array}{l} b := y \ \ /\!/ 1 \\ x := b \end{array} \qquad \text{(RPacq)}$$

The weakening of PS presented in §4.1 disallows the annotated behavior. Thread 1 cannot promise $y := 1$ because its certification has to execute a non-promised *acquire* RMW reading from an existing message against the capped memory; and also it cannot promise the RMW $z := 1$ before $y := 1$ because its certification requires reading $x := 1$. Nevertheless, as for RP, a global analysis may notice that z is accessed only by one thread and perform register promotion, yielding the annotated outcome. (Similarly, ARMv8 allows the annotated behavior of the corresponding target program.)

We note that the standard (Java) optimization of removing locks used by only one thread requires to perform register promotion on local locations accessed by acquire RMWs. Indeed, lock acquisitions are essentially acquire RMWs.

So, how can we allow such behaviors without harming DRF-RA? Our idea here is to enhance PS by allowing one to declare which thread will win the competition to perform an RMW reading from a given message m. Once such a declaration is made, RMWs performed by other threads cannot read from m.

The technical mechanism for these declarations is simple: we add a "reservation" step to PS, allowing a thread to reserve a timestamp interval that it plans to use later, without committing on how it will use it (what value and view will be picked). Once an interval is reserved, other threads are blocked from reusing timestamps in this interval. Intuitively, a reservation corresponds to promising the "read part" of the RMW, which confines the behavior of other threads. In particular, if a thread reserves an interval $(t_1, t_2]$ attached to

some message $(f, t_1]$, then other threads cannot read from the $(f, t_1]$ message with an RMW operation.

Since reservations are included in the machine memory (just like normal writes and promises), the semantics remains thread-local. Technically, reservations take the form $\langle x : (f, t] \rangle$ where $x \in \mathsf{Loc}$ and $(f, t]$ is a timestamp interval. To meet their purpose, we allow attaching reservations only immediately after existing concrete messages (f should be the "to"-timestamp of some existing message to the same location). Threads are also allowed to cancel their reservations (provided they can still certify their outstanding promises) if they no longer need to block an interval. This is technically needed for the soundness of register promotion (see [1, §B]).

Returning to the RPacq program above, reservations allow the annotated outcome. Thread 1 can first reserve the interval $(0, 1]$ for z. Then, it can promise $y := 1$ and certify its promise by using its own reservation to perform the RMW.

Intuitively, reservations are closer to the implementation of RMWs in ARM: reserving the read part of an RMW first and then writing the RMW at the reserved space later corresponds to execution of a load exclusive first and a (successful) write exclusive later.

Reservations are also used in the definition of the capped memory to fill the gaps between messages to the same location (§4.1). In the presence of reservations, however, the capped memory definition requires some care. First, the value of the cap messages $\widehat{m}_x$ should be the value of the maximal *concrete* message to x (reservations do not carry values). Second, when constructing the capped memory for thread i, if the maximal message to some location y is a reservation of thread i itself, then we do not add a cap message for y. In effect, during certification, the thread can execute any RMW on y but only after filling the reserved space on y. Other threads cannot execute an RMW on reservations of thread i, and so cannot interfere with respect to y.

4.3 Undefined Behavior

So far, we have described value-range optimizations by informally referring to a global analysis performed by the compiler. For our formal development, we introduce *undefined behavior* (UB). We note that UB, which is not supported in the original PS model, is also useful in a broader context (*e.g.*, to give sensible semantics to expressions like $x/0$).

In order to formally define global optimizations, we include in our language an abort instruction, **abort**, which causes UB. In turn, for a global invariant I (formally defined in §6.2), we allow the program transformation introducing at arbitrary program points the instruction **assert**(I), which is a syntactic sugar to **if** $\neg I$ **then abort**. This paves the way to further *local* optimizations, such as:

$$\begin{array}{l} \textbf{assert}(x \in \{0,1\}) \\ a := x \\ \textbf{if } a \in \{0,1\} \textbf{ then } c \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := x \\ c \end{array}$$

The standard semantics of UB is "catch-fire": UB should be thought as allowing any arbitrary sequence of operations. This enables common compiler optimizations (*e.g.*, **if** e **then** c **else abort** $\rightsquigarrow$ c). Nevertheless, to make sure the semantics is not overly weak, like any thread step, for taking an **abort**-step, the certification condition has to be satisfied (where the certifying thread may replace **abort** by any sequence of operations).

Our formal condition for taking an **abort**-step is somewhat simpler: we require that for every location x, the current view of the aborting thread for x should be lower than the "to"-timestamp of all the outstanding promises for x of that thread. We say a thread is *promise-consistent* when this condition is met. Recall that a thread can take a write step to a location x when the thread view of x is lower than the "to"-timestamp of the writing message. In turn, considering that taking an **abort**-step is capable of executing arbitrary write instructions, a thread is able to fulfill its outstanding promises when aborting if and only if it is promise-consistent.

4.4 Relaxed RMWs in Certifications

In PS 2.0, we opted to allow relaxed RMWs (that were non-promised before and read from a message that exists in the current memory) during certification of promises. This design choice can cause execution deadlocks:

$$\begin{array}{l} a := \textbf{FADD}(x, 1) \;\; /\!/ 0 \\ y := 1 + a \end{array} \;\Big\|\; b := \textbf{FADD}(x, 1) \qquad \text{(deadlock)}$$

Suppose that in the beginning of the run the thread 1 promises $y := 1$. This promise can be certified against the capped memory by reading from the cap message of x (whose value is 0). Now, thread 2 can perform its RMW, and block thread 1 from fulfilling its promise. Although allowing such deadlocks is awkward, they are inconsequential, since deadlocking runs are discarded from the definition of observable behavior.

Similarly, this choice enables somewhat dubious behaviors that seem to invalidate atomicity of relaxed RMWs: for instance, CDRF can have the annotated behavior if one **FADD** is made `rlx`. Such behaviors are actually unavoidable if one insists on allowing all (local and global) optimizations allowed by PS 2.0 ([1, §C] provides an example).

A stronger alternative would be to disallow relaxed RMWs during certification unless they were promised before the certification, or they read from a message that is added to the memory during certification. This can be easily achieved by defining the capped memory (against which threads certify their promises) to include a reservation instead of a cap message, which disallows to read from cap messages during certification. The resulting model is deadlock-free and it supports all (global and local) optimizations supported by PS 2.0, except for the local reordering of a relaxed RMW followed by a write. To see this consider the following example:

$$\begin{array}{l} a := \textbf{FADD}(x, 1) \;\; /\!/ 1 \\ y := 1 \end{array} \;\Big\|\; \begin{array}{l} b := y \;\; /\!/ 1 \\ x := b \end{array} \qquad \text{(LB-RMW)}$$

To read the annotated values, the run must start with thread 1 promising $y := 1$. Such a promise can only be certified if we allow relaxed RMWs that read an existing message during certification. Nevertheless, reordering the two instructions in thread 1 clearly exhibits the annotated behavior. In particular, since ARMv8 performs such reorderings, the mapping to ARMv8 should always include a dependency from relaxed RMWs, thereby incurring some (probably small) overhead.

5 Formal Model

In this section, we present our formal model, called PS 2.0, which combines and makes precise the ideas outlined above. For simplicity, we omit some features that were included in PS (plain accesses, fences, release sequences, and split and lower of promises).[7] All of these features are handled just like in PS and are included in our Coq formalization. The full operational semantics and the programming language are presented in [1, §A].

To keep the presentation simple and abstract, we do not fix a particular programming language syntax, and rather assume that the thread semantics is already provided as a labeled transition system, with transition labels `Silent` for a silent thread transition with no memory effect, $\mathtt{R}(o, x, v)$ for reads, $\mathtt{W}(o, x, v)$ for writes, $\mathtt{U}(o_\mathrm{r}, o_\mathrm{w}, x, v_\mathrm{r}, v_\mathrm{w})$ for RMWs, `Fail` for failing assertions, $\mathtt{Sys}(v)$ for a system calls.

The $o, o_\mathrm{r}, o_\mathrm{w}$ variables denote access modes, which can be either `rlx` or `ra`. We use `ra` for both release and acquire, and include two access modes in RMW labels: a read mode and a write mode. These naturally encode the syntax of the examples we discussed above, *e.g.*,

$$\textbf{FADD} \to \mathtt{U}(\mathtt{rlx}, \mathtt{rlx}, \ldots) \qquad \textbf{FADD}^{\mathtt{acq}} \to \mathtt{U}(\mathtt{ra}, \mathtt{rlx}, \ldots)$$
$$\textbf{FADD}^{\mathtt{acqrel}} \to \mathtt{U}(\mathtt{ra}, \mathtt{ra}, \ldots) \qquad \textbf{FADD}^{\mathtt{rel}} \to \mathtt{U}(\mathtt{rlx}, \mathtt{ra}, \ldots)$$

Next, we present the components of the PS 2.0 model.

Time. Time is a set of *timestamps* that is totally and densely ordered by $<$ with a minimum value, denoted by 0.

Views. A *view* is a function $V : \mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$. We use $\bot$ and $\sqcup$ to denote the natural bottom elements and join operations for views (pointwise extensions of the timestamp 0 and max operation on timestamps).

Concrete Messages. A *concrete message* takes the form $m = \langle x : v@(f, t], R\rangle$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $f, t \in \mathsf{Time}$, and $R \in \mathsf{View}$, such that $f < t$ or $f = t = 0$, and $R(x) \le t$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, $m.\mathtt{from}$, $m.\mathtt{to}$, and $m.\mathtt{view}$ the components of m.

Reservations. A *reservation* takes the form $m = \langle x : (f, t]\rangle$, where $x \in \mathsf{Loc}$, and $f, t \in \mathsf{Time}$ such that $f < t$. We denote by $m.\mathtt{loc}$, $m.\mathtt{from}$, and $m.\mathtt{to}$ the components of m.

[7] In particular, note that the system calls in this simplified model do not enforce sequentially consistent fences.

Messages. A *message* is either a concrete message or a reservation. Two messages m_1 and m_2 are *disjoint*, denoted by $m_1 \# m_2$, if they have different locations or disjoint timestamp intervals:

$$m_1 \# m_2 \triangleq m_1.\mathtt{loc} \neq m_2.\mathtt{loc} \lor m_1.\mathtt{to} < m_2.\mathtt{from} \lor m_2.\mathtt{to} < m_1.\mathtt{from}$$

Two sets M_1 and M_2 of messages are disjoint, denoted by $M_1 \# M_2$, if $m_1 \# m_2$ for every $m_1 \in M_1$ and $m_2 \in M_2$.

Memory. A *memory* is a (nonempty) pairwise disjoint finite set of messages. We write $M(x)$ for the sub-memory $\{\, m \in M \mid m.\mathtt{loc} = x \,\}$ and $\widetilde{M}$ for the set $\{\, m \in M \mid m = \langle \_ : \_@(\_, \_], \_\rangle \,\}$ of concrete messages in M.

Memory Operations. A memory M supports the *insertion* for a message m denoted by $M \xleftarrow{\mathrm{A}} m$ and given by $M \cup \{m\}$. It is only defined if: (i) $\{m\} \# M$, (ii) if m is a concrete message with $m.\mathtt{loc} = x$, then no message $m' \in M(x)$ has $m'.\mathtt{from} = m.\mathtt{to}$, and (iii) if m is a reservation with $m.\mathtt{loc} = x$, then there is some concrete message $m' \in \widetilde{M}(x)$ such that $m'.\mathtt{to} = m.\mathtt{from}$. Note that the second condition enforces that once a message is not an RMW (*i.e.*, its "from"-timestamp is not attached to another message), it never becomes an RMW (*i.e.*, its "from"-timestamp remains detached). Technically, this condition is required for the soundness of the register promotion.

Closed View. Given a view V and a memory M, we write $V \in M$ if, for every $x \in \mathsf{Loc}$, we have $V(x) = m.\mathtt{to}$ for some concrete message $m \in \widetilde{M}(x)$.

Thread States. A *thread state* is a triple $TS = \langle \sigma, V, P\rangle$, where σ is a local state, V is a thread view, and P is a memory. We denote by $TS.\mathtt{st}$, $TS.\mathtt{view}$, and $TS.\mathtt{prm}$ the components of a thread state TS.

Thread Configuration Steps. A *thread configuration* is a pair $\langle TS, M\rangle$, where TS is a thread state and M is a memory. We use $\bot$ as a thread configuration after a failure.

Fig. 2 presents the full list of thread configuration steps, which we discuss now. To avoid repetition, we use the helpers READ-HELPER and WRITE-HELPER. In these helpers, $\{x@t\}$ denotes the view assigning t to x and 0 to other locations.

PROMISE. A thread can take a PROMISE-step by adding a concrete message m to the set of outstanding promises P and update the memory M to $M \xleftarrow{\mathrm{A}} m$.

RESERVE and **CANCEL**. These two steps are specific to PS 2.0 model. In a RESERVE-step a thread reserves a timestamp interval by adding it to both the memory M and the set of outstanding promises $TS.\mathtt{prm}$. The thread is allowed to drop the reservation from the set of outstanding promises and the memory using the CANCEL-step.

READ. In this step a thread reads the value of a location x from a message $m \in M$ and extend its view. Following the READ-HELPER, the thread's view of location x is extended to

Memory Helpers:

(MEMORY: NEW)

$$\frac{}{\langle P, M\rangle \xrightarrow{m} \langle P, M \xleftarrow{\text{A}} m\rangle}$$

(MEMORY: FULFILL)

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

Thread Helpers:

(READ-HELPER)

$$\frac{m = \langle x : \_@(\_, t], R\rangle \in M \qquad V(x) \le t \qquad o = \texttt{rlx} \Rightarrow V' = V \sqcup \{x@t\} \qquad o = \texttt{ra} \Rightarrow V' = V \sqcup \{x@t\} \sqcup R}{\langle V, M\rangle \xrightarrow{o, m}_{\text{R}} \langle V', M\rangle}$$

(WRITE-HELPER)

$$\frac{m = \langle x : \_@(\_, t], R\rangle \qquad V(x) < t \qquad V' = V \sqcup \{x@t\} \qquad o = \texttt{rlx} \Rightarrow R = \bot \qquad o = \texttt{ra} \Rightarrow P(x) = \emptyset \wedge R = V' \qquad \langle P, M\rangle \xrightarrow{m} \langle P', M'\rangle}{\langle V, P, M\rangle \xrightarrow{o, m}_{\text{W}} \langle V', P', M'\rangle}$$

Thread Steps:

(PROMISE)

$$\frac{m = \langle \_ : \_@(\_, \_], R\rangle \qquad M' = M \xleftarrow{\text{A}} m \qquad R \in M'}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \to \langle\langle \sigma, \mathcal{V}, P \cup \{m\}\rangle, M'\rangle}$$

(RESERVE)

$$\frac{m = \langle \_ : (\_, \_]\rangle \qquad M' = M \xleftarrow{\text{A}} m}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \to \langle\langle \sigma, \mathcal{V}, P \cup \{m\}\rangle, M'\rangle}$$

(CANCEL)

$$\frac{m = \langle \_ : (\_, \_]\rangle \in P}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \to \langle\langle \sigma, \mathcal{V}, P \setminus \{m\}\rangle, M \setminus \{m\}\rangle}$$

(READ)

$$\frac{\sigma \xrightarrow{\text{R}(o, x, v)} \sigma' \qquad m = \langle x : v@(\_, \_], \_\rangle \qquad \langle V, M\rangle \xrightarrow{o, m}_{\text{R}} \langle V', M\rangle}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P\rangle, M\rangle}$$

(WRITE)

$$\frac{\sigma \xrightarrow{\text{W}(o, x, v)} \sigma' \qquad m = \langle x : v@(\_, \_], \_\rangle \qquad \langle V, P, M\rangle \xrightarrow{o, m}_{\text{W}} \langle V', P', M'\rangle}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P'\rangle, M'\rangle}$$

(UPDATE)

$$\frac{\sigma \xrightarrow{\text{U}(o_{\text{r}}, o_{\text{w}}, x, v_{\text{r}}, v_{\text{w}})} \sigma'' \qquad m_{\text{r}} = \langle x : v_{\text{r}}@(\_, t], \_\rangle \qquad m_{\text{w}} = \langle x : v_{\text{w}}@(t, \_], \_\rangle \qquad \langle V, M\rangle \xrightarrow{o_{\text{r}}, m_{\text{r}}}_{\text{R}} \langle V', M\rangle \qquad \langle V', P, M\rangle \xrightarrow{o_{\text{w}}, m_{\text{w}}}_{\text{W}} \langle V'', P'', M''\rangle}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma'', V'', P''\rangle, M''\rangle}$$

(SILENT)

$$\frac{\sigma \xrightarrow{\texttt{Silent}} \sigma'}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \to \langle\langle \sigma', \mathcal{V}, P\rangle, M\rangle}$$

(SYSTEM CALL)

$$\frac{\sigma \xrightarrow{\texttt{Sys}(v)} \sigma' \qquad P = \emptyset}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow{\texttt{Sys}(v)} \langle\langle \sigma', \mathcal{V}, P\rangle, M\rangle}$$

(FAILURE)

$$\frac{\sigma \xrightarrow{\texttt{Fail}} \bot \qquad \langle \sigma, \mathcal{V}, P\rangle \text{ is promise-consistent}}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow{\texttt{Fail}} \bot}$$

Machine Steps:

(MACHINE NORMAL)

$$\frac{\langle \mathcal{TS}(i), M\rangle \to^{+} \langle TS', M'\rangle \qquad \langle TS', M'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

(MACHINE SYSTEM CALL)

$$\frac{\langle \mathcal{TS}(i), M\rangle \to^{*} \xrightarrow{\texttt{Sys}(v)} \langle TS', M'\rangle \qquad \langle TS', M'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M\rangle \xrightarrow{\texttt{Sys}(v)} \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

(MACHINE FAIL)

$$\frac{\langle \mathcal{TS}(i), M\rangle \to^{*} \xrightarrow{\texttt{Fail}} \bot}{\langle \mathcal{TS}, M\rangle \xrightarrow{\texttt{Fail}} \bot}$$

Figure 2. Formal operational semantics

timestamp t. When the read is an acquire read, the view is also updated by the message view R.

WRITE and **UPDATE**. The write and the update steps cover two cases: a fresh write to memory (MEMORY:NEW) and a fulfillment of an outstanding promise (MEMORY:FULFILL). When a thread writes a message m with location x along with timestamp $(\_, t]$, t extends the thread's view of location x to memory M. A release write step additionally ensures that the thread has no outstanding promise on location x. Moreover, a release write attaches the updated thread view V' to the message m. The update step is similar, except that it first reads a message with a timestamp interval $(\_, t]$, and then, writes a message with an interval $(t, \_]$.

SILENT. A thread takes a SILENT-step to perform thread-local computation which updates only the local thread state.

SYSTEM CALL. A thread takes a SYSTEM CALL-step that emits an event with the call's input and output values.

FAILURE. We only allow a thread configuration $\langle TS, M\rangle$ to fail if TS is *promise-consistent*:

$$\forall m \in TS.\texttt{prm}, TS.\texttt{view}(m.\texttt{loc}) \le m.\texttt{to}$$

Cap View and Messages. The last message of a memory M to a location x, denoted by $\overline{m}_{M,x}$, is given by:

$$\overline{m}_{M,x} \triangleq \arg\max_{m \in M(x)} m.\texttt{to}$$

The *cap view* of a memory M, denoted by $\widehat{V}_M$, is given by:

$$\widehat{V}_M \triangleq \lambda x.\ \overline{m}_{\widetilde{M},x}.\texttt{to}$$

By definition, we have $\widehat{V}_M \in M$. The *cap message* of a memory M to a location x, denoted by $\widehat{m}_{M,x}$, is given by:

$$\widehat{m}_{M,x} = \langle x : \overline{m}_{\widetilde{M},x}.\texttt{val}@(\overline{m}_{M,x}.\texttt{to}, \overline{m}_{M,x}.\texttt{to} + 1], \widehat{V}_M\rangle$$

Capped Memory. The *capped memory* of a memory M with respect to a set of promises P, denoted by $\widehat{M}_P$, is an extension of M, constructed in two steps:

1. For every $m_1, m_2 \in M$ with $m_1.\texttt{loc} = m_2.\texttt{loc}$, $m_1.\texttt{to} < m_2.\texttt{to}$, and there is no message $m' \in M(m_1.\texttt{loc})$ such that $m_1.\texttt{to} < m'.\texttt{to} < m_2.\texttt{to}$, we include a reservation $\langle m_1.\texttt{loc} : (m_1.\texttt{to}, m_2.\texttt{from}]\rangle$ to $\widehat{M}_P$.
2. We include a cap message $\widehat{m}_{M,x}$ in $\widehat{M}_P$ for every location x unless $\overline{m}_{M,x}$ is a reservation in P.

Consistency. A thread configuration $\langle TS, M\rangle$ is called *consistent* if there exist TS', M' such that:

$$\langle TS, \widehat{M}_{TS.\texttt{prm}}\rangle \to^* \langle TS', M'\rangle \ \wedge \ TS'.\texttt{prm} = \emptyset$$

Machine steps. A *machine state* is a pair $\mathbf{MS} = \langle \mathcal{TS}, M\rangle$ consisting of a function $\mathcal{TS}$ assigning a thread state to every thread, and a memory M. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state σ_i^0, the $\bot$ thread view (all timestamps are 0), and an empty set of promises; and the initial memory M^0 consisting of one message $\langle x : 0@(0,0], \bot\rangle$ for each location x. The three possible machine steps are given at the bottom of Fig. 2. We use $\bot$ as a machine state after a failure.

Behaviors. To define what is externally observable during executions of a program P, we use the system calls that P's executions perform. More precisely, every execution induces a sequence of system calls, and the set of behaviors of P, denoted Beh(P), consists of all such sequences induced by executions of P. When a `Fail` occurs during the execution, Beh(P) consists of the sequence of system calls performed before the failure followed by an arbitrary sequence of system calls (reflecting an *undefined behavior*).

6 Results

We next present the results of PS 2.0. Except for Theorems 6.6 to 6.8 (whose proofs are given in [1]), all other results are fully mechanized in the Coq proof assistant. These results hold for the *full* model defined in [1, §A], not only for the simplified fragment presented in §5.

6.1 Thread-Local Optimizations

A transformation $P_{\text{src}} \rightsquigarrow P_{\text{tgt}}$ is *sound* if it does not introduce behaviors under any (parallel and sequential) context:

$$\forall\, C,\ \text{Beh}(C[P_{\text{src}}]) \supseteq \text{Beh}(C[P_{\text{tgt}}])\ .$$

PS 2.0 allows *all* compiler transformations supported by PS. Additionally, it supports replacing **abort** by arbitrary code (more precisely, $\mathbf{abort}; C_1 \rightsquigarrow C_2$). Since $\mathbf{assert}(e)$ is defined as $\mathbf{if}\ \neg e\ \mathbf{then\ abort}$, the following transformations are valid:

1. $\mathbf{assert}(e); C \rightsquigarrow \mathbf{assert}(e); C[\texttt{true}/e]$
2. $\mathbf{assert}(e) \rightsquigarrow \mathbf{skip}$

Thanks to thread-locality of PS and PS 2.0, we proved a theorem that combines and lifts the local simulation relations (almost without any reasoning on certifications) between pairs of threads S_i, T_i into a global simulation relation between the composed programs $S_1 \parallel \ldots \parallel S_n$ and $T_1 \parallel \ldots \parallel T_n$. This theorem allows us to easily prove soundness of the thread-local transformations using sequential and thread-local simulation relations. See Kang [11] and our Coq formalization for more details.

6.2 Value-Range Optimizations

First, we provide a global value-range analysis and prove its soundness in PS 2.0. A *value-range analysis* is a tuple $A = \langle J, S_1, \ldots, S_n\rangle$, where $J \in \text{Loc} \to \mathcal{P}(\text{Val})$ represents a set of possible values for each location and $S_i \subseteq \text{State}_i$ a set of possible local states of the underlying language (*i.e.*, excluding the thread views) for each thread i. The analysis is *sound* for a program P if (i) the initial value for each location is in J and the initial state of each thread i in P is in S_i; (ii) taking a step from each state in S_i necessarily leads to a state in S_i assuming that it only reads a value in J and guaranteeing that it only writes a value in J.

Now, we show that sound analysis for P holds in every reachable state of P.

Theorem 6.1 (Soundness of Value-Range Analysis)**.** *For a sound value-range analysis $\langle J, S_1, \ldots, S_n\rangle$ for P, if $\langle \mathcal{TS}, M\rangle$ is a reachable machine state for P, then $\mathcal{TS}(i).\texttt{st} \in S_i$ for every thread i, and $m.\texttt{val} \in J(x)$ for every $m \in \widetilde{M}(x)$.*

Second, we prove the soundness of global optimizations based on sound value-range analysis. An optimization based on a value-range analysis $A = \langle J, S_1, \ldots, S_n\rangle$ can be seen as inserting $\mathbf{assert}(e)$ at positions in thread i when e is always evaluated to `true`. For this, we define a relation, global_opt($A, P_{\text{src}}, P_{\text{tgt}}$), which holds when P_{tgt} is obtained from P_{src} by inserting valid assertions based on A.

Theorem 6.2 (Soundness of Global Optimizations)**.** *For a sound value-range analysis A of P_{src}, and for P_{tgt} such that* global_opt(A, P_{src}, P_{tgt})*, we have* $\text{Beh}(P_{src}) \supseteq \text{Beh}(P_{tgt})$.

6.3 Register Promotion

We prove soundness of register promotion. We denote by promote(s, x, r) the statement obtained from a statement s by promoting the accesses to memory location x to accesses to register r (see [1, §D] for a formal definition).

Theorem 6.3 (Soundness of Register Promotion)**.** *For a program $s_1 \| \ldots \| s_n$, if memory location x is only accessed by s_i (i.e., not occurring in s_j for every $j \neq i$) and register r is fresh in s_i (i.e., not occurring in s_i), we have:*

$$\text{Beh}(s_1 \| \ldots \| s_n) \supseteq \text{Beh}(s_1 \| \ldots \| \text{promote}(s_i, x, r) \| \ldots \| s_n)\ .$$

6.4 DRF Theorems

We prove four DRF theorems for PS 2.0: DRF-Promise, DRF-RA, DRF-Lock-RA and DRF-Lock-SC. First, we need several definitions:

- *Promise-free* (PF) semantics is the strengthening of PS 2.0 obtained by revoking the ability to make promises or reservations.

- *Release-acquire* (RA) is the strengthening of PF obtained by interpreting all memory operations as if they have `ra` access mode.
- *Sequential consistency* (SC) is the strengthening of RA obtained by forcing every read of a location x to read from the message with location x with the maximal timestamp and every write to a location x to write a message at a timestamp higher than any other x-message.

In the absence of promises, PS and PS 2.0 coincide:

Theorem 6.4. PF *is equivalent to the promise-free fragment of* PS*, and thus the same holds for* RA *and* SC.

We say that a machine state is `rlx`-race-free, if whenever two different threads may take a non-promise step accessing the same location and at least one of them is writing, then both are `ra` accesses.

Theorem 6.5 (DRF-Promise)**.** *If every* PF*-reachable machine state for P is* `rlx`*-race-free, then* $\mathrm{Beh}_{\mathsf{PF}}(P) = \mathrm{Beh}_{\mathsf{PS\,2.0}}(P)$.

This theorem is one of the key results of DRF theorems for PS 2.0. In our Coq formalization, we proved a stronger version of DRF-Promise, which is presented in [1, §E].

Theorem 6.6 (DRF-RA)**.** *If every* RA*-reachable machine state for P is* `rlx`*-race-free, then* $\mathrm{Beh}_{\mathsf{RA}}(P) = \mathrm{Beh}_{\mathsf{PS\,2.0}}(P)$.

Thanks to Theorems 6.4 and 6.5, the proof of DRF-RA for PS 2.0 is identical to that for PS given in [12].

Our DRF-Lock theorems given below generalize those for PS given in [12] in two aspects: our **Lock** are implemented with an *acquire* CAS rather than *acquire-release* CAS that was assumed in [12]; and our results cover **tryLock**, not just **Lock** and **Unlock**.

We define **tryLock**, **Lock** and **Unlock** as follows:

$$\begin{array}{rcl} a := \mathbf{tryLock}(L) & \triangleq & a := \mathbf{WCAS}^{\mathsf{acq}}(L, 0, 1) \\ \mathbf{Lock}(L) & \triangleq & \mathbf{do}\ a := \mathbf{tryLock}(L)\ \mathbf{while}\ !a \\ \mathbf{Unlock}(L) & \triangleq & L^{\mathtt{rel}} := 0 \end{array}$$

where $\mathbf{WCAS}^o$ is the weak CAS operation, which can either return **true** after successfully performing $\mathbf{CAS}^o$, or return **false** after reading *any* value from L with *relaxed* mode.

We prove DRF-Lock-RA and DRF-Lock-SC for programs using the three lock operations. We say such a program is *well-locked* if (1) locations are partitioned into *lock* and *non-lock* locations, (2) lock locations are accessed only by the three lock operations, and (3) **Unlock** is executed only when the thread holds the lock.

Theorem 6.7 (DRF-Lock-RA)**.** *For a well-locked program P, if every* RA*-reachable machine state for P is* `rlx`*-race-free for all non-lock locations, then* $\mathrm{Beh}_{\mathsf{RA}}(P) = \mathrm{Beh}_{\mathsf{PS\,2.0}}(P)$.

Theorem 6.8 (DRF-Lock-SC)**.** *For a well-locked program P, if every* SC*-reachable machine state reachable for P is race-free for all non-lock locations, then* $\mathrm{Beh}_{\mathsf{SC}}(P) = \mathrm{Beh}_{\mathsf{PS\,2.0}}(P)$.

The proofs of these theorems are given in [1, §F].

6.5 Compilation Correctness

Following Podkopaev et al. [19], we prove the correctness of mapping from PS 2.0 to hardware models (x86-TSO, POWER, ARMv7, ARMv8, RISC-V) using the Intermediate Memory Model, IMM, from which intended compilation schemes to the different architectures are already proved to be correct.

Theorem 6.9 (Correctness of Compilation to IMM)**.** *Every outcome of a program P under* IMM *is also an outcome of P under* PS 2.0*, i.e.,* $\mathrm{Beh}_{\mathsf{PS\,2.0}}(P) \supseteq \mathrm{Beh}_{\mathsf{IMM}}(P)$.

We note that this result (which is mechanized in Coq) requires the existence of a control dependency from the read part of each RMW operation. Such dependency exists "for free" in **CAS** operations, since its write operation (a store-conditional instruction) is anyway control-dependent on the read operation (a load-link instruction). However, when compiling **FADD**s to ARMv8, the compiler has to place "fake" control dependencies to meet this condition (and be able to use our theorem). We conjecture that a slightly more efficient compilation (standard) scheme of **FADD**s that does not introduce such dependencies is also sound. We leave this proof to a future work. In any case, our result is better than the one for PS by Podkopaev et al. [19] that requires an extra barrier ("ld fence") when compiling RMWs to ARMv8.

Remark 3. As in ARMv8, our compilation result to RISC-V uses release/acquire accesses. These accesses are not a part of RISC-V ISA, but the RISC-V memory model (RVWMO) is "*designed to be forwards-compatible with the potential addition*" of them [24, §14.1].

7 Related Work

We have already discussed the challenges in defining a 'sweet-spot' for a programming language concurrency model, which is neither too weak (*i.e.*, it provides programmability guarantees) nor too strong (*i.e.*, it allows efficient compilation). Java was the first language, where considerable effort was put into defining such a formal model [16], but the model was found to be flawed in that it did not permit a number of desired transformations [21]. To remedy this, C/C++ introduced a very different model based on 'per-execution' axioms [3], which was also shown to be inadequate [2, 13, 22, 23]. More recently, PS [12], which has already been discussed at length, addressed this challenge using the idea of locally certifiable promises. PS 2.0 improves PS by supporting global optimizations and better compilation of RMWs to ARMv8. We note that the promise-free fragment of PS 2.0 is identical to the promise-free fragment of PS.

Besides PS, there are three other approaches based on event structures [7, 8, 18]. Pichon-Pharabod and Sewell [18] defined an operational model based on plain event structures. Execution starts with a structure representing all possible program execution paths, and proceeds either by committing a prefix of the structure or by transforming it in a way

that imitates a compiler optimization (*e.g.*, by reordering accesses). The model also has a speculation step, whose aim is to capture transformations based on global value range analysis, but has side-condition that is rather difficult to check. The main downside of this model is its complexity, which hinders the formal development of results about it.

Jeffrey and Riely [8] defined a rather different model based on event structures, which constructs an execution via a two player game. The player tries to justify all the read events of an execution, while the opponent tries to prevent him. At each step, the player can extend the justified execution by one read event, provided that for any continuing execution chosen by the opponent, there is a corresponding write that produced the appropriate value. The basic model does not allow the reordering of independent reads, which means that compilation to ARM and Power are suboptimal. Although the model was later revised to fix the reordering problem [9], optimal compilation to hardware remains unresolved. Moreover, it does not support global optimizations and/or elimination of overwritten stores, since it forbids the annotated outcome of LB-G (in §1).

Chakraborty and Vafeiadis [7] introduced WEAKESTMO, a model based on *justified* event structures, which are constructed in an operational fashion by adding one event at a time provided it can be justified by already existing events. Justified event structures are then used to extract consistent executions, which in turn determine the possible outcomes of a program. While WEAKESTMO resolve PS's ARMv8 compilation problem [17], it does not formally support global optimizations. Moreover, WEAKESTMO does not support a class of strengthening transformations such as $\mathtt{W_{rel}} \rightsquigarrow \mathtt{F_{rel}}; \mathtt{W_{rlx}}$. Both PS and PS 2.0 support these transformations.

More recently, Java has been extended with different access modes in JDK 9 [14, 15]. Bender and Palsberg [4] formalized this extension with a 'per-execution' axiomatic model similar to RC11 [13]. The model disallows load-store reordering (LB behaviors) for atomic accesses, while allowing out-of-thin-air values for plain accesses. Because of the latter, global value analysis is unsound in this model. It remains unclear, however, whether transformations based on such (unsound) analysis might be sound or not.

8 Conclusion

We have presented PS 2.0, the first model that formally enables transformations based on global analysis while supporting programmability (via DRF guarantees and soundness of value-range reasoning) and efficient compilation (including various compiler thread-local optimizations). The inherent tension between these desiderata, together with our goal to have a thread-local small-step operational semantics, naturally leads to a rather intricate model, which is less abstract than alternative declarative models. Nevertheless, we note that PS 2.0, like its predecessor PS, is modeling weak behaviors with just two principles: (i) "views" for out-of-order execution of reads; and (ii) "promises" for out-of-order execution of writes. The added complexity of PS 2.0 is twofold: reservations and capped memory. We view reservations as a simple and natural addition to the promises mechanism. Capped memory is less natural and more complex. Fortunately, it is only a part of the certification process and not of normal execution steps. In addition, the DRF-Promise (and the other DRF theorems as well, Theorems 6.5 to 6.8) are methods to simplify the semantics. Programmers may safely use the PF or the RA fragment of PS 2.0, which has only views (without any promises, certifications, reservations, or capped memory), when their programs are avoiding data race via release-acquire and lock synchronization.

We also note that PS 2.0 allows some seemingly dubious behaviors, such as *"read from unexecuted branch"* [5]:

$$\begin{array}{l|l} a := x \;\; /\!/ 42 & b := y \;\; /\!/ 42 \\ y := a & \textbf{if } b = 42 \\ & \textbf{then } \; x := b \\ & \textbf{else } \; x := 42 \end{array} \qquad \text{(RFUB)}$$

The annotated behavior is allowed in PS 2.0 (as in PS and C/C++11). Aiming to support local compiler optimizations, this is actually unavoidable. Practical compilers (including gcc and llvm) may observe that thread 2 writes 42 to x regardless of which branch is taken, and optimize the program of thread 2 to $b := y; x := 42$ (such optimization is a "trace-preserving transformation" [12]). The resulting program is essentially the LB program (see §1), whose annotated behavior can be obtained by mainstream architectures.

Finally, to the best of our knowledge, PS 2.0 supports all practical compiler optimizations performed by mainstream compilers. As a future goal, we plan to extend it with sequentially consistent accesses (backed up with DRF-SC guarantee) and C11-style consume accesses.

Acknowledgments

We thank the PLDI'20 reviewers for their helpful feedback. Chung-Kil Hur is the corresponding author. Sung-Hwan Lee, Minki Cho, and Chung-Kil Hur were supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-53. Anton Podkopaev was supported by JetBrains Research and RFBR (grant number 18-01-00380). Ori Lahav was supported by the Israel Science Foundation (grant number 5166651), by Len Blavatnik and the Blavatnik Family foundation, and by the Alon Young Faculty Fellowship.

References

[1] 2020. *Coq development and supplementary material for this paper.* http://sf.snu.ac.kr/promising2.0

[2] Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *POPL 2013*. 235–248.

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

[4] 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

[5] Hans-Juergen Boehm. 2019. P1217R2: Out-of-thin-air, revisited, again. wg21.link/p1217 [Online; accessed 22-March-2020].

[6] C/C++11 mappings to processors 2019. Retrieved July 3, 2019 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[7] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL, Article 70 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290383

[8] Alan Jeffrey and James Riely. 2016. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *LICS 2016*. ACM, 759–767.

[9] 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

[10] JMM causality test cases 2019. Retrieved November 17, 2019 from http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html

[11] Jeehoon Kang. 2019. *Reconciling low-level features of C with compiler optimizations.* Ph.D. Dissertation. Seoul National University.

[12] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017* (Paris, France). ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

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

[14] Doug Lea. 2019. JEP 188: Java Memory Model Update. Retrieved November 17, 2019 from http://openjdk.java.net/jeps/188

[15] Doug Lea. 2019. Using JDK 9 Memory Order Modes. Retrieved November 17, 2019 from http://gee.cs.oswego.edu/dl/html/j9mm.html

[16] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL 2005*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

[17] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2019. Reconciling Event Structures with Modern Multiprocessors. *arXiv preprint arXiv:1911.06567* (2019).

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

[19] 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

[20] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[21] Jaroslav Sevcik and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. 27–51.

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

[23] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, New York, NY, USA, 867–884. https://doi.org/10.1145/2509136.2509532

[24] Andrew Waterman and Krste Asanović. 2017. The RISC-V Instruction Set Manual Volume I: User-Level ISA. https://riscv.org/specifications/isa-spec-pdf/