Grounding Thin-Air Reads with Event Structures

SOHAM CHAKRABORTY, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

The key challenge in defining the concurrency semantics of a programming language is how to enable the most efficient compilation to existing hardware architectures, and yet forbid programs from reading thin-air values, i.e., ones that do not appear in the program. At POPL'17, Kang et al. achieved a major breakthrough by introducing the 'promising' semantics that came with results showing that it was a good candidate solution to the problem. Unfortunately, however, the promising semantics is rather complicated, and due to its complexity it contains some flaws and limitations that are very hard to address.

In response, we present an alternative solution to this problem based on event structures. We show that it is indeed a solution by establishing the standard results about the semantics (DRF theorems, implementation and optimization correctness) as well as a formal connection to the semantics of Kang et al. Further, we show that it is easier to adapt, by extending the semantics to cover features (such as SC accesses) that are not supported by Kang et al. and to rule out some dubious behaviors admitted by the promising semantics.

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

Additional Key Words and Phrases: Weak memory consistency, concurrency, promising semantics, C/C++11

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

1 INTRODUCTION

By now, relaxed memory consistency has become the de facto concurrency paradigm for architectures, languages, and compilers. Yet, there is still no agreed adequate definition of the concurrency semantics of a language like C or Java. Developing such a definition is very challenging because it must be strong enough to eliminate undesirable behaviors and provide certain programming guarantees, and yet weak enough to allow efficient compilation—to support compiler optimizations and optimal mappings to the mainstream architectures.

We demonstrate the conflicting requirements with an example. Assume that initially $X = Y = 0$ and all shared memory accesses are C11 'relaxed' atomic accesses [ISO/IEC 14882 2011; ISO/IEC 9899 2011]. The question is whether the outcome $a = b = 1$ should be allowed.

$$\begin{array}{l} a = X; \\ \textbf{if}\,(a) \\ \quad Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ \textbf{if}\,(b) \\ \quad X = 1; \end{array} \quad \text{(CYC)} \qquad \begin{array}{l} a = X; \\ Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ \textbf{if}\,(b) \\ \quad X = 1; \end{array} \quad \text{(LB)} \qquad \begin{array}{l} a = X; \\ \textbf{if}\,(a)\ Y = 1; \\ \textbf{else}\ Y = 1; \end{array} \Bigg\| \begin{array}{l} b = Y; \\ \textbf{if}\,(b) \\ \quad X = 1; \end{array} \quad \text{(LBfd)}$$

In the CYC program, this outcome is considered an "out-of-thin-air" (OOTA) behavior that should be forbidden. Allowing it would violate the data-race-freedom (DRF) property [Adve and Hill 1990],

Authors' addresses: Soham Chakraborty, MPI-SWS, Germany, sohachak@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2019/1-ART70
https://doi.org/10.1145/3290383

which states that programs without races under *sequential consistency* (SC) [Lamport 1979] should not have any non-SC behaviors. Indeed, under SC, CYC is race-free and has only the outcome $a = b = 0$. In contrast, the outcome $a = b = 1$ should be allowed for LB because it arises when the compiler or the hardware reorders the (clearly independent) instructions of the first thread. This outcome should also be allowed for LBfd because it may be converted into LB by an optimizing compiler. As noted by Batty et al. [2015], these programs cannot be correctly distinguished by "per execution" models (e.g., C/C++11 [Batty et al. 2011], Linux kernel model [Alglave et al. 2018], and all the hardware models).

To solve this problem, a number of alternative modeling approaches have emerged [Jeffrey and Riely 2016; Kang et al. 2017; Manson et al. 2005; Pichon-Pharabod and Sewell 2016]. The most complete among these approaches is the *promising semantics* (PS) of Kang et al. [2017], who also claim to have resolved the conflicting requirements. Their model forbids the mentioned OOTA behavior in CYC, provides DRF guarantees, and supports the expected program transformations and mappings to x86-TSO and PowerPC. Although PS is a major step towards providing a formal model for C11 concurrency, as with the other approaches, it suffers from certain problems, which we discuss in detail in §7. For instance, PS's intended compilation scheme for read-modify-write instructions to ARMv8 is unsound because PS forbids the weak behavior of the FADD program shown in Fig. 6, while it is allowed by ARMv8. To address these problems, we tried to adapt the PS definition but found its definition style to be inflexible: any small change affected nearly all the results about the semantics.

In this paper, we develop an alternative approach to formalize the semantics of concurrent programs. Following Jeffrey and Riely [2016], we use *event structures* [Winskel 1986], a formalism that captures multiple executions of a program in one structure without referring to the concrete program syntax. We adapt them to the weak consistency setting by introducing in §2 the notions of *justified* and *visible* events. Event structures differentiate between the CYC and LB programs, but not between LB and LBfd, which enables us to give the right semantics to these programs.

In our experience, event structures are equally powerful (in that we establish essentially the same results as Kang et al. [2017]) but much more flexible than PS. To demonstrate their flexibility, we present two variants of our model, both of which allow the expected weak behavior of FADD.

The first variant is the WEAKEST ("*weak e*vent *st*ructure") model. In §4, we prove that it is weaker than PS, which allows us to leverage the existing compilation results about PS. Because of this, however, WEAKEST allows certain dubious program behaviors (e.g., that of Coh-CYC in Fig. 3).

The second variant, WEAKESTMO, avoids such undesirable behaviors by maintaining a per-location *modification order*, which totally orders any two non-conflicting writes to the same location. Consequently, WEAKESTMO is strictly stronger than WEAKEST and incomparable to PS. Moreover, it meets the established criteria for a good memory model:

- It disallows *OOTA behavior*—at least for the established Java causality tests (§3.5).
- It provides *DRF guarantees*, which allow programmers to write concurrent code defensively and reason about their code without needing to understand the WEAKESTMO definition (§5).
- It supports *efficient compilation*. We prove the correctness of the mapping from C11 to WEAKESTMO (§6.1), source-to-source optimizations within WEAKESTMO (§6.2), and mappings to x86 and PowerPC (§6.3). For simplicity, in the case of PowerPC, we establish the correctness of a suboptimal mapping with a spurious conditional branch after each read.
- It supports *all* types of memory accesses, fences, and memory orders in C/C++11 except 'consume' atomics, which are known to be problematic.

In the following, we start in §2 with an informal overview of our approach. From §3 onwards, we give the formal details. The proofs can be found in our appendix [Chakraborty and Vafeiadis 2018].

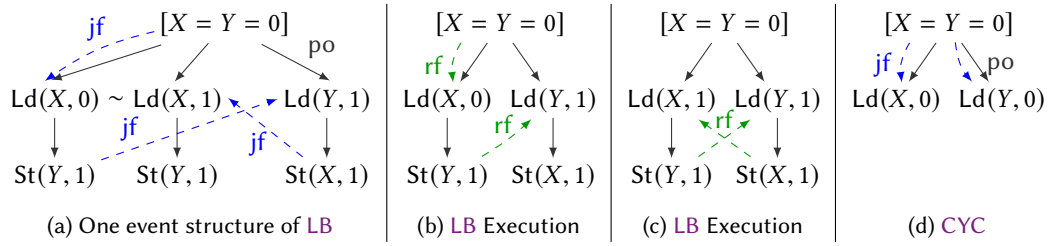


Fig. 1. One event structure of LB, two executions extracted from it, and the only event structure of CYC.

2 MAIN IDEAS

In this overview section, we gradually explain our event structure models (§2.1-§2.3) and compare them with the promising semantics of Kang et al. [2017] (§2.4).

2.1 Event Structures as a Solution to the OOTA Problem

We define the semantics of programs in two steps. First, we iteratively construct a "consistent" event structure from the program and then we extract its "consistent" executions.

An event structure is a representation of multiple executions of a program. It comprises a set of events representing the program's individual memory accesses and two relations over these events:

- The *program order* (po) represents the control flow order of the program; it relates two events a and b if they belong to the same thread and a is executed before b.
- The *conflict relation* (cf) relates events of the same thread that *cannot* belong to the same execution. A conflict, for example, occurs between two load events correspond to the same program instruction that return different values. By definition, cf is a symmetric relation. Moreover, when a and b conflict, and c is after b in program order, then a and c also conflict. That is, a conflict results in two different branches of the event structure, after which all events conflict with one another. For conciseness, in diagrams (e.g., in Fig. 1a), we display only the first conflicting edges (a.k.a. immediate conflicts) with ~ and omit the induced ones.

To these basic event structures, we add an extra relation, which we call the *justified-from* relation (jf). For each read event in the event structure, jf associates a (prior) write event that *justifies* the read returning its value. Namely, the write should have written the same value to the same location that the read is reading from.

An event structure is constructed by appending one event at a time following the program text. For a write event to be added to the structure, all its po-previous events must have been added; for a read event, there must additionally already exist a justifying write event. Thus, by construction $(\mathrm{po} \cup \mathrm{jf})$ is acyclic. The construction terminates when we can no longer add a new event to the structure. In general, whenever we add an event to an event structure, we check that the resulting structure is *consistent* (i.e., whether it satisfies a few consistency conditions that will be defined later) and discard any event structures that do not satisfy those conditions. Among other things, consistency ensures that sequential programs have their expected semantics by forbidding, for instance, a read to be justified by a stale write—one overwritten by a more recent write to the same location po-before the read.

As an example, Fig. 1a displays one consistent event structure of LB. Note that we cannot construct this event structure for program CYC, because once we add the $\mathrm{Ld}(X, 0)$ and $\mathrm{Ld}(Y, 0)$ events as shown in Fig. 1d, no further events can be added.

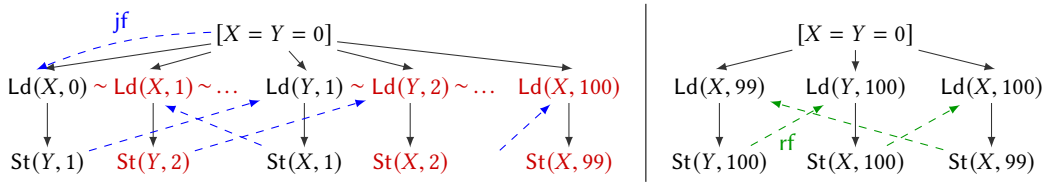


Fig. 2. A naively constructed event structure of RNG and C11-consistent extracted execution from it.

Next, given a consistent event structure, we extract the set of its maximal conflict-free subsets, which we call *executions*. (These are called configurations by Winskel [1986].) As an example (ignoring the rf edges for the moment), Figs. 1b and 1c show the two executions that can be extracted from the event structure in Fig. 1a. We filter the set of extracted executions by a certain execution consistency predicate, discarding those executions that are inconsistent. The allowed outcomes of a program are determined by these consistent executions.

One standard consistency requirement for executions is that every read should get its value from some write to the same location. This is formalized by a *reads-from* (rf) edge from the write to the read. Other consistency axioms may further constrain the allowed reads-from edges.

In this paper, we take execution consistency predicate to be that of RC11 (i.e., the repaired definition of C11 by Lahav et al. [2017]) without its $\mathsf{po} \cup \mathsf{rf}$ acyclicity requirement. We thus get a model that is *stronger* than C11 (which allows OOTA) and *weaker* than RC11 (which forbids load-store reordering). In particular, it assigns the right meaning to CYC, LB, and LBfd.

In general, due to different ways in which events may be appended to an event structure, a program may have multiple consistent event structures, and each such event structure may yield one or more consistent executions.

2.2 A Problem with the Simple Construction Scheme and How to Solve it

Although the simple scheme presented so far works well for LB and its variants, it suffers from a subtle problem demonstrated by the following "random number generator" litmus test.

$$Y = X + 1; \;\Big\|\; X = Y; \;\Big\|\; \begin{array}{l}\textbf{if}\,(X == 100)\\ \quad X = 99;\end{array} \qquad \text{(RNG)}$$

First, as shown in Fig. 2, even for this obviously terminating program, the event structure construction can go on for ever. Thread 1 reads $X = 0$ and writes $Y = 1$; thread 2 then reads $Y = 1$ and writes $X = 1$. Then, in a conflicting execution branch, thread 1 reads $X = 1$ and writes $Y = 2$; thread 2 reads this value and writes $X = 2$. Then, thread 1 can read $X = 2$ and the process can be repeated indefinitely. This problem is not so serious because one can always stop the construction at some arbitrary point. The more serious problem is that the construction can generate an event structure containing the $\mathsf{St}(X, 99)$ event of the last thread. From there, we can extract the execution shown at the right part of Fig. 2, which is consistent according to C11 (as witnessed by the depicted rf edges) and leads to a very counterintuitive outcome.

Clearly, the straightforward approach of constructing an event structure does not work here. In this example, we might blame the addition of $\mathsf{Ld}(X, 1)$, as it is the first event that will never appear in any reasonable run of the program. From then on all other events added (shown in red in Fig. 2) are equally bogus. We may observe that $\mathsf{Ld}(X, 1)$ is $(\mathsf{po} \cup \mathsf{jf})^+$-reachable from $\mathsf{Ld}(X, 0)$ in the event structure and thus the creation of $\mathsf{Ld}(X, 1)$ *causally depends* on its conflicting event $\mathsf{Ld}(X, 0)$. As the two events are in conflict, they cannot both occur in a single execution, so perhaps a possible constraint is that *no event should causally depend on a conflicting event.*

Unfortunately, this restriction turns out to be too strong because it also rules out the $a = b = 1$ outcome of LB (cf. Fig. 1a). We therefore relax the restriction by introducing the notion of *visibility*, and adapt the extraction of executions from event structures to *discard any executions containing invisible events*. As we will shortly see, all events of Fig. 2 drawn in red are invisible, and so the problematic extracted execution shown to right of the figure is discarded.

To define the notion of visible events, we introduce the *equal-write* (ew) relation, which relates two writes on the same location with the same written value that are in conflicting execution branches. For instance, in Fig. 1a, the two $\mathsf{St}(Y, 1)$ events are ew-related, whereas in Fig. 2 no events are ew-related.

Given this relation, we say that an event e is *invisible* whenever there is a conflicting write event in its $(\mathsf{po} \cup \mathsf{jf})^+$-prefix that does not have an equal write in the same execution branch as e (i.e., $(\mathsf{po}^? \cup \mathsf{po}^{-1})$-related to e). For example, the $\mathsf{Ld}(X, 1)$ event is invisible in Fig. 2 because of the $\mathsf{St}(Y, 1)$ store, whereas $\mathsf{Ld}(X, 1)$ is visible in Fig. 1a. Note that by definition if an event is invisible in an event structure, then so are all its po-later events.

The equal-write relation also allows to make a formal connection between the *justification* relation (jf) at the level of event structures and the *reads-from* relation (rf) at the level of extracted executions. The idea is to also define rf at the level of event structures in terms of jf and ew. We say that a read event r *reads from* a write w if it is justified by w or by one of its equal writes. When extracting an execution from an event structure, we take their rf relations to match for the extracted events. (We invite the reader to check that this is indeed so for the executions in Fig. 1).

2.3 Two Options Regarding Coherence: The WEAKEST and WEAKESTMO Models

We move on to *coherence*, a basic property enforced by almost all memory models.[1] Coherence states that programs containing only one shared variable exhibit only SC behaviors. To ensure this property, axiomatic memory models require the existence of a (strict) total order among all writes to a given location. We call the union of all these per-location orders the *modification order* (mo).

With our event structure model, a natural question arises: should we enforce coherence at the level of event structures or only at the level of extracted executions? In the following, we will consider both options. We call WEAKEST the plain model that checks coherence only at the execution level, while WEAKESTMO the model that records a modification order in its event structures and discards any event structures containing an incoherent execution.

To make this concrete, consider the following standard coherence litmus test program, whose annotated outcome should be forbidden.

$$\begin{array}{l|l} X = 1; & X = 2; \\ a = X; \ // \ 2 & b = X; \ // \ 1 \end{array} \tag{Coh}$$

At the execution level, the outcome $a = 2 \wedge b = 1$ is forbidden because mo must order two writes to X. Suppose without loss of generality that $\mathsf{St}(X, 1)$ is ordered before $\mathsf{St}(X, 2)$. Then, returning $b = 1$ is inconsistent because it reads from a write that is overwritten by a write before the read.

At the event structure level, WEAKEST allows the following event structure:

(Coh-ES)

This, on its own, is not a problem because we cannot extract from it a consistent execution, and so WEAKEST forbids the incoherent outcome.

[1] The exceptions are Itanium and Java, which enforce a slightly weaker coherence property for their plain accesses.

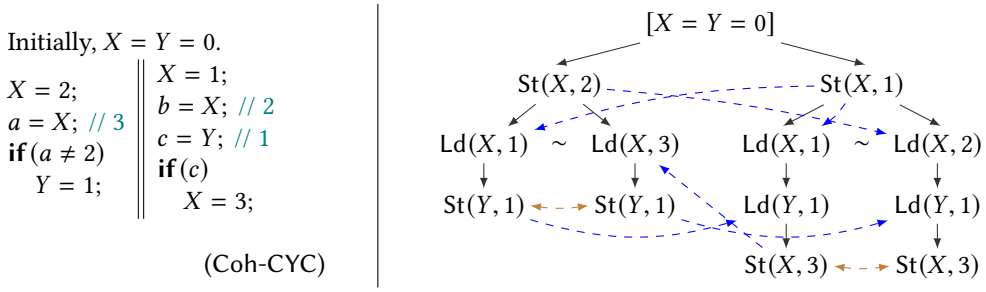


Fig. 3. A program and its WEAKEST event structure yielding the outcome $a = 3 \wedge b = 2 \wedge c = 1$.

WEAKESTMO goes further and disallows the incoherent event structure as well. Similar to executions, WEAKESTMO event structures have a modification order, mo. The model requires mo to totally order non-conflicting writes to the same location, and declares event structures containing coherence violations as inconsistent.

We move on to a more complex example that distinguishes the two models. Consider the program shown in Fig. 3 and the outcome $a = 3 \wedge b = 2 \wedge c = 1$. We note that this outcome is rather dubious: one can see it as arising due to a violation of coherence or a circular dependency. Indeed, if $b = 2$ then $X = 1$ must be mo-before $X = 2$, and so the first thread cannot read $a = 1$. Likewise, it should not be able to read $a = 3$ because that depends on $c = Y$ reading 1, which circularly depends on the first thread reading $a = 3$. We further note that this outcome is not observable on any machine and the program cannot be transformed in any reasonable fashion so as to enable that outcome. (In particular, read-after-write elimination ruins the outcome, and the only reordering possible is moving the $c = Y$ instruction earlier, which does not enable any further optimizations.)

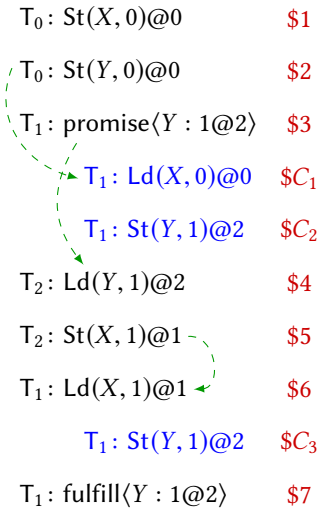
This outcome is, however, allowed by the WEAKEST model because of the event structure shown in Fig. 3. In contrast, it is not allowed by WEAKESTMO because the displayed event structure is incoherent. Recall that under WEAKESTMO, non-conflicting stores will have to be ordered by mo. If $\mathsf{St}(X, 2)$ is ordered before $\mathsf{St}(X, 1)$, then the $\mathsf{Ld}(X, 2)$ load violates coherence. If, instead, $\mathsf{St}(X, 2)$ is ordered after $\mathsf{St}(X, 1)$, then the $\mathsf{Ld}(X, 1)$ load of the first thread is incoherent.

As we will shortly see, this outcome is also allowed by the promising semantics (PS) [Kang et al. 2017]. In fact, we will show that WEAKEST can indeed simulate PS, which allows us to leverage the existing results about PS. In contrast, since the WEAKESTMO model fails to simulate PS, we will separately establish its compilation correctness.

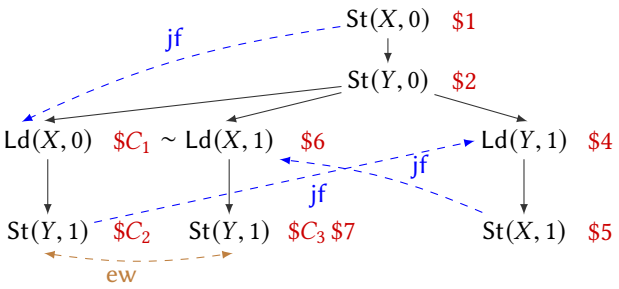
2.4 Relating WEAKEST and WEAKESTMO to the Promising Semantics

We start with a simplified informal presentation of PS sufficient for programs containing only relaxed loads and stores. (We refer the reader to Kang et al. [2017] for further details.) PS is defined operationally by a machine, whose state consists of the states of the threads and the shared memory. The shared *memory* is modeled as a set of messages, representing the writes that the program has performed. Messages are of the form $\langle x : v@t \rangle$ denoting a write of value v to location x at timestamp t. Timestamps enforce coherence by totally ordering the writes to each location.

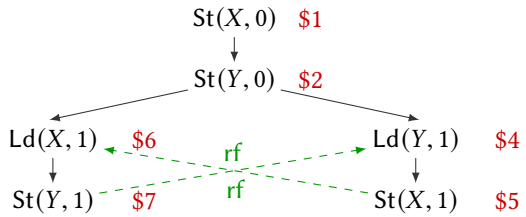
Each thread maintains a *view*, V, mapping each memory location to a timestamp, indicating the message with the maximum timestamp that the thread is aware of. When a thread performs a store $\mathsf{St}(x, v)$, it chooses an unused timestamp $t > V(x)$, adds the message $\langle x : v@t \rangle$ to memory, and updates its thread view of x to timestamp t. When it performs a load of x, it reads from a message $\langle x : v@t \rangle$ with $t \geq V(x)$ and updates its thread view of x to timestamp t.



(a) Promise steps of LB.



(b) The corresponding event structure in the WEAKEST model.



(c) The corresponding C11-consistent extracted execution.

Fig. 4. A promise machine execution of LB with its corresponding event structure and extracted execution.

Besides executing its next instruction, a thread can also *promise* to write value v at some location x at some future timestamp $t > V(x)$. It does so by appending the message $\langle x : v@t \rangle$ to memory as if the write were actually performed, and keeping track of the message as being an outstanding promise in its local state. To make such a promise, the respective thread must be able to *certify* its promise: namely, to be able to perform a write fulfilling the promise in a finite number of thread-local steps. More generally, PS requires that after each execution step all outstanding promises be certifiable. When a write step later fulfills an outstanding promise, it marks it as fulfilled.

Example 1 (Load buffering)**.** Figure 4a displays an execution of the promise machine for LB resulting in the outcome $a = b = 1$. The steps are as follows.
($1,$2) Initially, X and Y are zero and the memory is $\{\langle X : 0@0\rangle, \langle Y : 0@0\rangle\}$.
($3) Thread T_1 promises to write $Y = 1$ at timestamp 2 and appends a message $\langle Y : 1@2\rangle$ to the memory. The certification steps read from message $\langle X : 0@0\rangle$ (C_1) and fulfill the promise (C_2).
($4) T_2 executes $b = Y$; and reads from message $\langle Y : 1@2\rangle$.
($5) T_2 executes $X = 1$; adding message $\langle X : 1@1\rangle$ to the memory.
($6) T_1 executes $a = X$; reading message $\langle X : 1@1\rangle$. Note that T_1 can still fulfill its promise by C_3.
($7) Finally, T_1 executes the $Y = 1$; statement and fulfills its promise. □

We now demonstrate on the load buffering example how we can simulate the promise machine execution with our WEAKEST model. Figure 4b displays the corresponding event structure following the steps of the promise machine execution.

($1, $2) We add the two initialization stores $\mathsf{St}(X, 0)$ and $\mathsf{St}(Y, 0)$ at the beginning of the graph.
($3) For a promise step, we cannot immediately add a corresponding store to the event structure. We therefore look at its promise-free certificate execution (i.e., steps C_1 and C_2) and add the corresponding events to the structure. This adds events $\mathsf{Ld}(X, 0)$ and $\mathsf{St}(Y, 1)$.

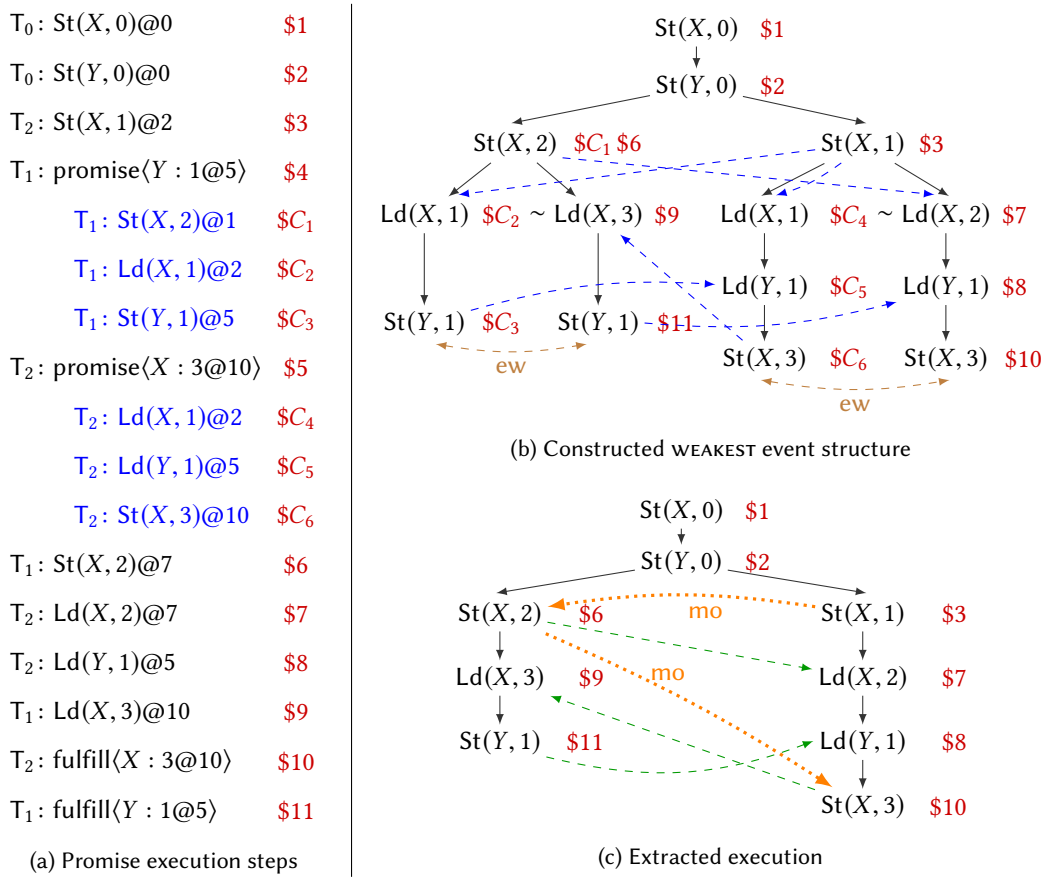


Fig. 5. A promise execution of Coh-CYC with its corresponding WEAKEST event structure and execution.

($4) In the promise machine, this step reads from the promised message $\langle Y : 1@2 \rangle$. Recall that the corresponding event in the event structure (i.e., $\$C_2$) was created by the certificate run. So we append a $\mathsf{Ld}(Y, 1)$ event justifying from $\$C_2$.
($5) We simply append $\mathsf{St}(X, 1)$ to the event structure.
($6) We construct event $\mathsf{Ld}(X, 1)$ justifying from $5 (i.e., the event corresponding to message $\langle X : 1@1 \rangle$). which is appended immediately after $\mathsf{St}(Y, 0)$ in the first thread. As $\mathsf{St}(Y, 0)$ already has an immediate po-following event in the first thread, $\mathsf{Ld}(X, 1)$ is in conflict with $\mathsf{Ld}(X, 0)$. Note that at this point, the newly added event $6 is *invisible* because of the conflicting store $\$C_2$ that is justified from thread T_2. To make $6 visible, we look at T_1's certification run and add the corresponding events to the event structure (i.e., event $\$C_3$). Since the certificate must fulfill all outstanding promises, it must contain a write producing the same message as that of step $\$C_2$ (namely, the write $\$C_3$). We relate the two writes by ew, which makes events $6 and $\$C_3$ visible.
($7) At this step, we would normally append a $\mathsf{St}(Y, 1)$ event immediately after event $6, but we notice that such an event already exists because of the previous step. We therefore do nothing and simply record that $7 corresponds to the same event as $\$C_3$.

By selecting the non-promised events, we can then extract the execution shown in Fig. 4c. The constraints about the messages' timestamps ensure that the execution is consistent.

Initially, $X = Y = Z = 0$.

$a = X$; // reads 1	‖	$c = Y$; // reads 1
$b = \text{fadd}(Z, a)$;		$X = c$;
$Y = b + 1$;		

(FADD)

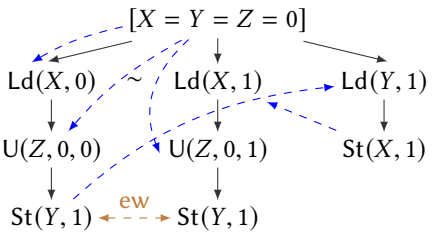


Fig. 6. The FADD program whose $a = c = 1$ outcome is allowed by ARMv8, WEAKEST, and WEAKESTMO. The instruction $\text{fadd}(Z, a)$ atomically increments the value at location Z by a, and returns the old value of Z.

Example 2 (Coh-CYC from Fig. 3)**.** Although PS associates totally ordered timestamps with its messages, interestingly it still allows Coh-CYC's dubious behavior as demonstrated in Fig. 5a. At execution step \$3 thread T_2 writes $X = 1$ at timestamp 2. To certify the promise after \$4, the certificate step $\$C_1$ chooses a smaller timestamp (i.e., 1) than that of \$3. In contrast, later at step \$6 it chooses a larger timestamp for the same write, which eventually leads to the dubious outcome.

Following the same construction as outlined previously, we can simulate the PS execution with the WEAKEST event structure shown in Fig. 5a. (This is the same event structure as the one shown in Fig. 3.) We can hence extract the execution shown in Fig. 5c, which witnesses PS's outcome. □

We move on to an example with atomic updates: the program FADD shown in Fig. 6. This is another variant of the load-buffering program, where the store to Y on thread T_1 depends on the result of a fetch-and-add instruction. The final value of Z therefore depends on the previous load of X, but the return value of fadd does not. As a result, the ARMv8 model [Pulte et al. 2018] allows the annotated weak outcome $a = c = 1$.

Both our models, WEAKEST and WEAKESTMO allow the same outcome with the event structure displayed in Fig. 6. The execution obtained by extracting the events of the second branch of T_1 and the events of T_2 is consistent and witnesses the discussed outcome.

Nevertheless, as we will shortly see, PS forbids this outcome. In order to handle atomic updates, PS has a few more features that we have not yet mentioned. Specifically, instead of a single timestamp, PS messages carry a timestamp range $(\text{from}, \text{to}]$. Generally, different messages in memory have disjoint timestamp ranges; so whenever two messages to Z have adjacent messages, this means that there cannot be another message to X in between, neither in the current memory nor in any future extension of that memory. This feature is used to satisfy the key invariant of atomic updates, namely that they read from their immediately preceding write in modification order. Further, to ensure that an execution never gets stuck because of an update whose slot is taken by another thread, PS has a more sophisticated certification condition. It requires that the outstanding promises of each thread be thread-locally certifiable not only in the present memory but also in *every future memory* (i.e., in every memory that extends the current one with possibly more messages).

Now consider running FADD under PS. To get the outcome $a = c = 1$, we must start with a promise. Clearly, the machine cannot promise $X = 1$ nor $Z = 1$ because these writes cannot be certified in the current memory. However, it also cannot promise $Y = 1$ in T_1 because there exists a future memory, namely one with the message $\langle Z : 42@(0, 1]\rangle$, where T_1 cannot fulfill its promise (it will write $Y = 43$). As a result, PS cannot produce the outcome $a = c = 1$, which in turn means that its intended compilation to ARMv8 is unsound. To restore soundness of compilation from PS to ARMv8, Podkopaev et al. [2019] insert a `dmb.ld` fence after every atomic update.

In summary, we have shown that WEAKEST is strictly weaker than both WEAKESTMO and PS, while WEAKESTMO and PS are incomparable. Although our models further correct the aforementioned

counterexample of PS's compilation to ARMv8, we do not have proof of soundness of compilation from WEAKESTMO to ARMv8.

3 FORMALIZATION: THE WEAKEST AND WEAKESTMO MODELS

In this section, we formally define the WEAKEST and WEAKESTMO models that were introduced in the previous section.

We take programs, $\mathbb{P}$, to be top-level parallel compositions of some number of threads, $T_1 \| \ldots \| T_N$. To make our semantics parametric over the programming language syntax, we represent each thread T_i as some non-empty set of traces denoting the possible sequences of memory accesses it can produce. Formally, a trace, τ, is a (finite) sequence of labels given by the following grammar:

$$lab ::= \mathsf{Ld}_{o_r}(x, v) \mid \mathsf{St}_{o_w}(x, v) \mid \mathsf{U}_{o_u}(x, v, v') \mid \mathsf{F}_{o_u} \quad \text{(Event labels)}$$

$$o_r ::= \text{NA} \mid \text{RLX} \mid \text{ACQ} \mid \text{SC} \quad \text{(Memory orders for reads)}$$

$$o_w ::= \text{NA} \mid \text{RLX} \mid \text{REL} \mid \text{SC} \quad \text{(Memory orders for writes)}$$

$$o_u ::= \text{RLX} \mid \text{ACQ} \mid \text{REL} \mid \text{ACQ-REL} \mid \text{SC} \quad \text{(Memory orders for updates and fences)}$$

We have load (Ld), store (St), update (U), and fence (F) labels. Updates are used to model the effect of atomic read-modify-write (RMW) instructions, such as fetch-and-add and compare-and-swap (CAS). The labels record the location accessed (x), the values read and/or written (v, v'), and the corresponding C11 *memory order* ($o_r/o_w/o_u$): non-atomic (NA), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (ACQ-REL), or sequentially consistent (SC). In increasing strength, these orders are: $\text{NA} \sqsubset \text{RLX} \sqsubset \{\text{ACQ}, \text{REL}\} \sqsubset \text{ACQ-REL} \sqsubset \text{SC}$.

We assume that the thread semantics is *prefix-closed*, *receptive*, and *deterministic*. Prefix-closedness requires for every $\tau{\cdot}\tau' \in T_i$ (where $\cdot$ denotes trace concatenation), we have $\tau \in T_i$. In particular, this means that the empty trace is always included in T_i. Receptiveness requires that whenever $\tau{\cdot}lab \in T_i$ and lab is a read label (a load or an update) of location x, then for every value v, there exists a read label lab' of location x reading that value v such that $\tau{\cdot}lab' \in T_i$. In other words, whenever a thread can read x, it can do so for any possible value. Determinism requires that whenever $\tau{\cdot}lab \in T_i$ and $\tau{\cdot}lab' \in T_i$, then the two labels agree except perhaps due to receptiveness. That is, if unequal, then they are both reads (loads or updates) of the same location.

Definition 1. An *event* is a tuple, $\langle id, tid, lab\rangle$, where $id \in \mathbb{N}$ is a unique identifier for the event, $tid \in \mathbb{N}$ identifies the thread to which the event belongs (we use $tid = 0$ for initialization events), and lab denotes the label of the event.

A *read event* is either a load or an update, whereas a *write event* is either a store or an update. Let $\mathcal{E}$ denote the set of all events, $\mathcal{R}$ the set of all read events, $\mathcal{W}$ the set of all write events, and $\mathcal{F}$ the set of fence events. We use subscripts to constraint those sets; e.g., $\mathcal{E}_{\sqsupseteq \text{REL}}$ denotes all events whose memory order is at least REL. We write $e.\mathsf{id}$, $e.\mathsf{tid}$, $e.\mathsf{lab}$, $e.\mathsf{op}$, $e.\mathsf{loc}$, $e.\mathsf{ord}$, $e.\mathsf{rval}$, and $e.\mathsf{wval}$, to return (when applicable) the identifier, thread, label, type (Ld, St, U, F), location, memory order, read value, and written value respectively.

Relational Notation. Let $R, S \subseteq \mathcal{E} \times \mathcal{E}$ be binary relations on events. We write $R\,;S$ for the relational composition of R and S. Let R^{-1} denote the inverse of R, $\mathsf{dom}(R)$ its domain and $\mathsf{codom}(R)$ its range. We write $R^?$, R^+, R^*, $R^=$ for the reflexive, the transitive, the reflexive-transitive, and the reflexive-symmetric closures of R respectively. (Reflexive closure is with respect to the set $\mathcal{E}$, while reflexive-symmetric closure means $R^= \triangleq (R \cup R^{-1})^?$.) Further, $R|_{\mathsf{loc}} \triangleq \{(e, e') \in R \mid e.\mathsf{loc} = e'.\mathsf{loc}\}$ restricts R on pairs of events of the same location. Similarly, $R|_{\neq \mathsf{loc}} \triangleq R \setminus R|_{\mathsf{loc}}$ restricts R on pairs of events of different locations.

The $[A]$ notation denotes an identity relation on set A, i.e. $[A](x, y) \triangleq x = y \wedge x \in A$. For a finite set A that is totally ordered by R, we write $\text{sequence}_R(A)$ to denote the sequence of its elements ordered according to R.

Definition 2. An *event structure*, $G \triangleq \langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo} \rangle$, contains the following components:

- $\mathsf{E} \subseteq \mathcal{E}$ is the *set of events* of the event structure containing the set of initialization events, E_0, which has exactly one event with label $\mathsf{St}_{\text{NA}}(x, 0)$ for every location x. For a thread i, we write E_i to refer to the events of the event structure with thread identifier i.
- $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ denotes the *program order*. It is a strict partial order on events recording when an event precedes another one in the control flow of the program. Initialization events are po-before all non-initialization events (i.e., $\mathsf{E}_0 \times (\mathsf{E} \setminus \mathsf{E}_0) \subseteq \mathsf{po}$), and that non-initialization events related by po have the same thread identifier.
  Further, we assume that non-initialization po-predecessors of an event are totally ordered by po. That is, we never have distinct $a, b \in \mathsf{E} \setminus \mathsf{E}_0$ that have a common po-successor and are not po-related to one another. In relational notation, $[\mathsf{E} \setminus \mathsf{E}_0]; \mathsf{po}; \mathsf{po}^{-1}; [\mathsf{E} \setminus \mathsf{E}_0] \subseteq \mathsf{po}^{=}$.
  We note that po is not total over events of the same thread. In particular, we say that events of the same thread not related by the program order are in *conflict* with one another.
  $$G.\mathsf{cf} \triangleq \{(a, b) \in G.\mathsf{E} \times G.\mathsf{E} \mid a.\mathsf{tid} = b.\mathsf{tid} \neq 0\} \setminus \mathsf{po}^{=} \qquad \text{(Conflict relation)}$$
  By definition, the conflict relation, cf, is symmetric and irreflexive. Moreover, it is forward-closed with respect to the program order (i.e., $\mathsf{cf} ; \mathsf{po} \subseteq \mathsf{cf}$): if two events conflict, then so are all their future events.
- $\mathsf{jf} \subseteq \mathsf{E} \times \mathsf{E}$ is the *justified-from* relation, which relates a write event to the reads it justifies. That is, whenever $\mathsf{jf}(w, r)$, then $w \in \mathcal{W}$, $r \in \mathcal{R}$, $w.\mathsf{loc} = r.\mathsf{loc}$ and $w.\mathsf{wval} = r.\mathsf{rval}$ hold. We require that every read is justified from exactly one write. That is, jf^{-1} is functional and $\mathsf{dom}(\mathsf{jf}) = \mathsf{E} \cap \mathcal{R}$.
- $\mathsf{ew} \subseteq [\mathcal{W}_{\sqsubseteq \text{RLX}}] ; \mathsf{cf}|_{\mathsf{loc}} ; [\mathcal{W}_{\sqsubseteq \text{RLX}}]$ is the *equal-writes* relation, a symmetric relation relating conflicting (relaxed) writes on the same location writing the same value.
- $\mathsf{mo} \subseteq ((\mathsf{E} \cap \mathcal{W}) \times (\mathsf{E} \cap \mathcal{W}))|_{\mathsf{loc}}$ is the *modification order*: a strict partial order that orders write operations on the same memory location. We require that mo is total on non-conflicting writes to the same location and that equal writes have the same mo-successors, i.e., $G.\mathsf{ew}; G.\mathsf{mo} \subseteq \mathsf{mo}$. (In the WEAKEST model, we do not use the mo component in the event structure.)

Given an event structure G, we use notation $G.X$ to project the X component out of G. When G is clear from the context, we sometimes omit the "$G.$".

We now define a number of derived relations on event structures. First, we define *synchronizes-with* (sw) and *happens-before* (hb) following RC11 [Lahav et al. 2017] (replacing C11's reads-from relation with justification). The sw definition is quite involved and states conditions under which a justification edge synchronizes two threads. The simplest case inducing synchronization is when an acquire read event reads from a release write. More advanced cases involve release/acquire fences and/or reads through sequences of updates (known as release sequences in C11). An event a *happens before* an event b if there is a path from a to b consisting of program order and synchronization edges.

$$G.\mathsf{sw} \triangleq [\mathcal{E}_{\sqsupseteq \text{REL}}] ; ([\mathcal{F}] ; G.\mathsf{po})^{?} ; [\mathcal{W}] ; G.\mathsf{po}|^{?}_{\mathsf{loc}} ; [\mathcal{W}_{\sqsupseteq \text{RLX}}] ; G.\mathsf{jf}^{+} ; [\mathcal{R}_{\sqsupseteq \text{RLX}}] ; (G.\mathsf{po} ; [\mathcal{F}])^{?} ; [\mathcal{E}_{\sqsupseteq \text{ACQ}}] \qquad \text{(Synchronizes-with)}$$

$$G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup G.\mathsf{sw})^{+} \qquad \text{(Happens-before)}$$

We say that two events are in *immediate conflict* ($\sim$) if neither of them has a po-previous event conflicting with the other event. Two events are in *extended conflict* (ecf) if they happen after

conflicting events, which means that they cannot be part of the same execution.

$$G.\sim \triangleq G.\mathsf{cf} \setminus (G.\mathsf{cf} \,;\, G.\mathsf{po} \cup G.\mathsf{po}^{-1} \,;\, G.\mathsf{cf}) \qquad \text{(Immediate conflict)}$$

$$G.\mathsf{ecf} \triangleq (G.\mathsf{hb}^{-1})^? \,;\, G.\mathsf{cf} \,;\, G.\mathsf{hb}^? \qquad \text{(Extended conflict)}$$

Next, the *reads-from* relation, rf, lifts the justified-from relation to all equal writes; i.e., whenever r is justified by w and w is equal to w' and w' and r do not conflict, then $\mathsf{rf}(w', r)$. This, in particular, means that rf^{-1} is not necessarily functional at the level of event structures.

$$G.\mathsf{rf} \triangleq (G.\mathsf{ew}^? \,;\, G.\mathsf{jf}) \setminus G.\mathsf{cf} \qquad \text{(Reads-from relation)}$$

The next two relations—*reads-before/from-read* (fr) and the *extended coherence order* (eco)—are also defined as in RC11 [Lahav et al. 2017]. A read event r reads before a write event w if r reads from some write w' that is mo-after w. (In the formal definition, we subtract the identity relation so as to avoid saying that an update reads before itself.) Finally, the extended coherence order is the transitive closure of the union of the three coherence-enforcing relations (rf, mo, and fr).

$$G.\mathsf{fr} \triangleq (G.\mathsf{rf}^{-1} \,;\, G.\mathsf{mo}) \setminus [\mathcal{E}] \qquad \text{(Reads-before/from-read relation)}$$

$$G.\mathsf{eco} \triangleq (G.\mathsf{rf} \cup G.\mathsf{mo} \cup G.\mathsf{fr})^+ \qquad \text{(Extended coherence order)}$$

Since the WEAKEST model does not record the modification order, we define $\mathsf{mo}_{\mathsf{strong}}$, a stronger version of the modification order, which relates writes to the same location that are ordered by happens-before. We then use $\mathsf{mo}_{\mathsf{strong}}$ in place of mo to derive stronger versions of fr and eco.

$$G.\mathsf{mo}_{\mathsf{strong}} \triangleq [\mathcal{W}] \,;\, G.\mathsf{hb}|_{\mathsf{loc}} \,;\, [\mathcal{W}] \qquad \text{(Strong modification order)}$$

$$G.\mathsf{fr}_{\mathsf{strong}} \triangleq (G.\mathsf{rf}^{-1} \,;\, G.\mathsf{mo}_{\mathsf{strong}}) \setminus [\mathcal{E}] \qquad \text{(Strong reads-before)}$$

$$G.\mathsf{eco}_{\mathsf{strong}} \triangleq (G.\mathsf{rf} \cup G.\mathsf{mo}_{\mathsf{strong}} \cup G.\mathsf{fr}_{\mathsf{strong}})^+ \qquad \text{(Strong extended coherence order)}$$

Remark 1. Note that the definitions above allow ew to relate a relaxed store with a conflicting relaxed update writing the same value to the same location. This flexibility is not needed for any of our proofs about WEAKESTMO; in fact, it would suffice for ew to relate only events with the same label and, in the case of updates, with the same justification (jf). Under these restrictions, then sw as defined above is equivalent to the C11-definition of sw that uses rf in the place of jf. Without these restrictions, however, the definition with rf is problematic because adding a new update event and making it ew-related to an existing store could induce synchronizations between existing events in the event structure. The only reason why we did not restrict ew is so that we can establish that WEAKEST is weaker than the promising semantics in §4. The promising semantics, in particular, allows certifying a promised message with an update event and then fulfilling it with a write, and vice versa.

3.1 Event Structure Consistency Checking

We say that an event e is visible in an event structure G if all the writes recursively used to justify e that are in conflict with e have some equal write that does not conflict with e. Formally:

Definition 3. An event e is *visible* in an event structure G if $e \in G.\mathsf{E}$ and

$$[\mathcal{W}] \,;\, (G.\mathsf{cf} \cap G.\mathsf{jfe}; (G.\mathsf{po} \cup G.\mathsf{jf})^*; G.\mathsf{jfe}; G.\mathsf{po}^?) \,;\, [\{e\}] \subseteq G.\mathsf{ew} \,;\, G.\mathsf{po}^=$$

where $G.\mathsf{jfe} \triangleq G.\mathsf{jf} \setminus G.\mathsf{po}$ denotes the *external justification edges*. We write $\mathsf{vis}(G)$ for the set of all visible events of an event structure G.

As we will shortly see, our model essentially treats extended conflicts as normal conflicts.

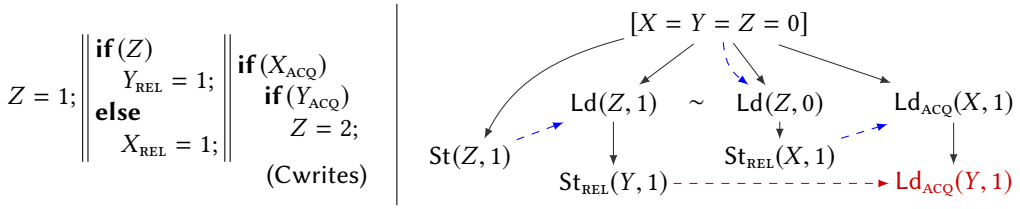


Fig. 7. Constraint (CF) restricts $Z = 2$ in the Cwrites program.

Definition 4. An event structure G is *consistent* according to model $M \in \{\textsc{weakest}, \textsc{weakestmo}\}$, written $\mathsf{isCons}_M(G)$, iff the following conditions hold:

- (CF) No event can be in extended conflict with itself: $G.\mathsf{ecf}$ is irreflexive.
- (CFJ) A write cannot justify a read in extended conflict: $G.\mathsf{jf} \cap G.\mathsf{ecf} = \emptyset$.
- (VISJ) Only visible events justify reads of other threads: $\mathsf{dom}(G.\mathsf{jfe}) \subseteq \mathsf{vis}(G)$.
- (ICF) Immediately conflicting events must be reads: $G.{\sim} \subseteq \mathcal{R} \times \mathcal{R}$.
- (ICFJ) Immediately conflicting reads cannot be justified by the same or by equal writes: $G.\mathsf{jf}\,;\, G.{\sim}\,;\, G.\mathsf{jf}^{-1}\,;\, G.\mathsf{ew}^?$ is irreflexive.
- (COH) if $M = \textsc{weakest}$, then $G.\mathsf{hb}\,;\, G.\mathsf{eco}^?_{\mathsf{strong}}$ is irreflexive.
- (COH′) if $M = \textsc{weakestmo}$, then $G.\mathsf{hb}\,;\, G.\mathsf{eco}^?$ is irreflexive.

We now explain these constraints in order.

The first constraint (CF) ensures that every event in the event structure could belong to some execution by checking that its hb-predecessors are conflict-free. This, in particular, ensures that po-related reads do not observe values from two different execution branches of some other thread in the case where both reads-from edges would result in a synchronization. We explain the scenario with the Cwrites program in Fig. 7 taken from Chakraborty and Vafeiadis [2017]. Intuitively, this program should never produce the outcome $Z = 2$, as there is no execution under which both X and Y could have the value 1. With this in mind, we would like to rule out the event structure shown in Fig. 7 that contains an execution branch where both acquire loads of X and Y read the value 1. It is easy to see that the (CF) does not hold in the event structure, as $\mathsf{St}_{\mathrm{REL}}(Y, 1)$ conflicts with $\mathsf{St}_{\mathrm{REL}}(X, 1)$ which happens before $\mathsf{Ld}_{\mathrm{ACQ}}(Y, 1)$.

The second constraint (CFJ) forbids a read to be justified from a write event in extended conflict. Intuitively, since conflicting events can never appear in a single execution, allowing reads to read from conflicting events could generate OOTA results. This constraint rules out the problematic event structure for the variants of Cwrites where one of the reads of the third thread is relaxed.

Remark 2. While (CF) rules out the problematic event structure of Fig. 7, it is not strictly needed for rule out the $Z = 2$ behavior, because no RC11-consistent execution can be extracted from that event structure. Nevertheless, (CF) and (CFJ) are needed for a more complicated variant of Cwrites in Kang et al. [2017, Appendix A.2], where without them we cannot guarantee DRF-RA (Theorem 4).

The third constraint (VISJ) says that reads are either justified locally from the same thread or they are justified from some *visible* write. This constraint forbids the undesired event structure in Fig. 2 for the RNG program.

The next two constraints place some restrictions on immediate conflicts that rule out creating unnecessary duplication in the event structure. (ICF) requires that immediate conflicts are between read events. Because we assume that the thread semantics is deterministic, immediately conflicting events must be created by the same program instruction. If, for example, they originate from a

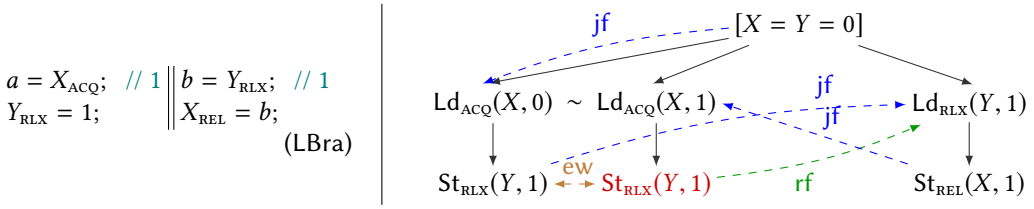
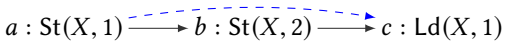


Fig. 8. Outcome $a = b = 1$ is not allowed in LBra because of a coherence violation.

load instruction, they will all be load events. If they originate for a compare-and-swap (CAS), they will either be update events (for the successful case) or load events (for the case when the CAS fails). In both of these cases, it may well make sense to have an immediate conflict in the event structure. In contrast, it does not make sense to create an immediate conflict from a store or a fence instruction, because both events will have the exact same label, which is why (ICF) rules out such events. Similarly, (ICFJ) disallows creating an immediate conflict by reading from the same write. That is, we cannot have two reads in immediate conflict that are justified by the same write or by equal writes. Again such reads will read the same value, and so will lead to duplication in the event structure.

Finally, the (COH) and (COH′) constraints enforce that the event structure is coherent at various levels. Both constraints ensure that the happens-before (hb) is irreflexive and that rf does not contradict hb. We explain the latter with the program in Fig. 8, which is another variant of the load buffering litmus test. In this case, the outcome $a = b = 1$ must be forbidden and is actually disallowed by C/C++11 (despite thin-air problems). The reason is that if $a = 1$, then the release write of X synchronizes with the acquire read of X, and so the read of Y would have to read from an hb-later write. Our coherence constraints similarly enforce that the event structure in Fig. 8 is inconsistent. More specifically, the event structure without the second $\mathrm{St}_{\mathrm{RLX}}(Y, 1)$ event (i.e., the one displayed in red) is consistent but adding that event along with the displayed ew relation renders the event structure inconsistent. The equal-writes relation together with Y's justification edge induce the displayed reads-from edge, which contradicts happens-before.

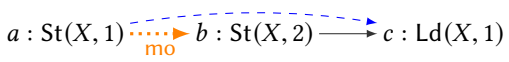
In addition, the coherence constraints also ensure that reads do not read overwritten values, as for example in the following inconsistent event structure:



(Basic coherence violation)

The justification edge induces an rf-edge from a to c, and in turn an $\mathrm{fr}_{\mathrm{strong}}$-edge from c to b, which contradicts hb.

The two coherence constraints differ only in the presence of concurrent writes to the same location; i.e., of two writes to a given location that are not ordered by hb. If, for example, in the event structure above, the events a and b were not related by po, WEAKEST would allow the justification edge. WEAKESTMO would also allow it but only if the mo-edge went from b to a. If mo went from a to b, as depicted below, the event structure would be inconsistent:



(WEAKESTMO coherence violation)

As already explained in §2.3, the difference is evident in programs Coh and Coh-CYC. Specifically, the event structure of Coh shown in that section (Coh-ES) is consistent according to the WEAKEST model, but not according to WEAKESTMO: whichever way $\mathrm{St}(X, 1)$ and $\mathrm{St}(X, 2)$ are ordered by

$$A \subseteq G.\mathsf{E}_{e.\mathsf{tid}} \qquad \mathsf{dom}([\mathsf{E}_{e.\mathsf{tid}}]\,;\mathsf{po}\,;[A]) \subseteq A \qquad \mathsf{labels}(\mathsf{sequence}_{\mathsf{po}}(A)) \cdot e.\mathsf{lab} \in \mathbb{P}(e.\mathsf{tid})$$
$$\mathsf{E}' = \mathsf{E} \uplus \{e\} \qquad \mathsf{po}' = \mathsf{po} \cup (A \times \{e\}) \qquad \mathsf{isCons}_M(\langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}'\rangle) \qquad CF = (\mathsf{E}_{e.\mathsf{tid}} \setminus A)$$
$$\textbf{if } e \in \mathcal{R} \textbf{ then } \exists w \in \mathsf{E} \cap \mathcal{W}.\ \mathsf{jf}' = \mathsf{jf} \cup \{(w,e)\} \wedge w.\mathsf{loc} = e.\mathsf{loc} \wedge w.\mathsf{wval} = e.\mathsf{rval} \textbf{ else } \mathsf{jf}' = \mathsf{jf}$$
$$\textbf{if } e \in \mathcal{W}_{\sqsubseteq \mathrm{RLX}} \textbf{ then } \mathsf{AddEW}(\mathsf{ew}, \mathsf{ew}', CF, e) \textbf{ else } \mathsf{ew}' = \mathsf{ew}$$
$$\textbf{if } e \in \mathcal{W} \textbf{ then } \mathsf{AddMO}(\mathsf{mo}, \mathsf{mo}', \mathsf{E}, CF, \mathsf{ew}, e) \textbf{ else } \mathsf{mo}' = \mathsf{mo}$$
$$\langle \mathsf{E}, \mathsf{po}, \mathsf{jf}, \mathsf{ew}, \mathsf{mo}\rangle \rightarrow_{\mathbb{P},M} \langle \mathsf{E}', \mathsf{po}', \mathsf{jf}', \mathsf{ew}', \mathsf{mo}'\rangle$$

Fig. 9. One construction step of a program's event structure, where $\mathsf{labels}(a_1 \cdots a_n) \triangleq a_1.\mathsf{lab} \cdots a_n.\mathsf{lab}$.

mo, there is a coherence violation. The event structure of Coh-CYC shown in Fig. 5b is similarly inconsistent according to WEAKESTMO, because it contains the incoherent Coh-ES as a substructure.

3.2 Event Structure Construction

Having defined event structures and their consistency, we move on to explain how they are constructed.

We start with the initial event structure G_{init} that contains as events only the initialization events, $\mathsf{E}_0 = \{a_1, \ldots, a_k\}$ where k is the number of variables in the program and each a_i has label $\mathsf{St}_{\mathrm{NA}}(X_i, 0)$ and thread identifier 0. All the other components of G_{init} are simply the empty relation.

Starting with G_{init}, we construct an event structure of a program $\mathbb{P}$ incrementally by adding one event at a time using the rule shown in Fig. 9. The reduction relation has the form $G \rightarrow_{\mathbb{P},M} G'$ and represents a transition from event structure G to event structure G' of the program $\mathbb{P}$ according to model $M \in \{\text{WEAKEST}, \text{WEAKESTMO}\}$.

The rule adds one new event e to the event structure G. To do so, it picks the set A of its po-predecessors from the same thread. This set A includes events of thread $e.\mathsf{tid}$ that are already in the event structure ($\mathsf{E}_{e.\mathsf{tid}}$), and is closed under program order prefix. That is, any po-predecessor of an event in A should also belong to A. Next, it checks that local thread semantics can produce the label of e, after the sequence of labels of A.

In the second line of the rule, the set of events is extended to include e, and the program order is similarly extended to place e after all the events in A. It checks that the resulting event structure is consistent according to model M where the updates to the event structure's remaining components will be defined shortly, and discards any inconsistent event structures. We finally let CF contain all the events conflicting with e, which are exactly those events from the same thread as e that were not placed before it.

The third line of the rule concerns the update of the justification relation. If e is a read event, then there must already be some write event w in the event structure that writes the value that e reads to the same location. In that case, the justification relation is extended with the edge (w, e). If e is not a read (i.e., it is a store or a fence event), then jf remains unchanged.

The last two lines concern the update of the equal-writes relation and the modification order. The former is updated whenever the new event is a relaxed write, whereas the latter is updated for all new write events. We use the following auxiliary definitions for updating ew and mo:

$$\mathsf{AddEW}(\mathsf{ew}, \mathsf{ew}', CF, e) \triangleq \exists W \subseteq \mathcal{W}_{\sqsubseteq \mathrm{RLX}} \cap CF.\ (\forall w \in W.\ w.\mathsf{loc} = e.\mathsf{loc} \wedge w.\mathsf{wval} = e.\mathsf{wval}) \wedge \mathsf{ew}' = \mathsf{ew} \cup (W \times \{e\}) \cup (\{e\} \times W)$$

$$\mathsf{AddMO}(\mathsf{mo}, \mathsf{mo}', \mathsf{E}, CF, \mathsf{ew}, e) \triangleq \exists w \in \mathcal{W} \cap \mathsf{E} \setminus CF.\ w.\mathsf{loc} = e.\mathsf{loc} \wedge \mathsf{mo}' = \mathsf{mo} \cup (\mathsf{dom}(\mathsf{mo}^?; \mathsf{ew}^?; [\{w\}]) \times \{e\}) \cup (\{e\} \times \mathsf{codom}([\{w\}]; \mathsf{mo}; \mathsf{ew}^?))$$

$$\mathsf{X.sw}_{\mathsf{C11}} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; \mathsf{X.po})^? ; [\mathcal{W}] ; \mathsf{X.po}|_{\mathsf{loc}}^? ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; \mathsf{X.rf}^+ ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (\mathsf{X.po} ; [\mathcal{F}])^? ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}]$$

$$\mathsf{X.hb}_{\mathsf{C11}} \triangleq (\mathsf{X.po} \cup \mathsf{X.sw}_{\mathsf{C11}})^+$$

$$\mathsf{X.scb} \triangleq \mathsf{X.po} \cup (\mathsf{X.po}|_{\neq \mathsf{loc}} ; \mathsf{X.hb}_{\mathsf{C11}} ; \mathsf{X.po}|_{\neq \mathsf{loc}}) \cup \mathsf{X.hb}_{\mathsf{C11}}|_{\mathsf{loc}} \cup \mathsf{X.mo} \cup \mathsf{X.fr}$$

$$\mathsf{X.psc}_{\mathsf{base}} \triangleq [\mathcal{E}_{\mathrm{sc}}] ; ([\mathcal{F}_{\mathrm{sc}}] ; \mathsf{X.hb}_{\mathsf{C11}})^? ; \mathsf{X.scb} ; (\mathsf{X.hb}_{\mathsf{C11}} ; [\mathcal{F}_{\mathrm{sc}}])^? ; [\mathcal{E}_{\mathrm{sc}}]$$

$$\mathsf{X.psc}_{\mathsf{F}} \triangleq [\mathcal{F}_{\mathrm{sc}}] ; (\mathsf{X.hb}_{\mathsf{C11}} \cup \mathsf{X.hb}_{\mathsf{C11}} ; \mathsf{X.eco} ; \mathsf{X.hb}_{\mathsf{C11}}) ; [\mathcal{F}_{\mathrm{sc}}]$$

Fig. 10. Additional definitions needed for execution consistency (taken from Lahav et al. [2017]).

AddEW selects a set W of conflicting relaxed writes to the same location and that write the same value as e. Then, ew is extended by relating e to all the elements in W. AddMO selects a write event w in the graph that writes to the same location as e and does not conflict with it. Since the event structure is initialized, such a write always exists. It then puts e immediately after w in mo.

3.3 Execution Extraction in the WEAKEST and WEAKESTMO Models

As mentioned in §2.1, after a consistent event structure is constructed, we extract from it a set of conflict-free subsets of events, which we call executions.

More formally, an execution X is a tuple of the form $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ whose components in order denote the set of events, the program order, the reads-from relation, and the modification order of the execution. As with event structures, we use the dot notation (e.g., X.E) to project the relevant components of an execution.

The set of events, the program order, and the reads-from relation have the same constraints as for event structures. For example, when event r reads from event w, then r is a read, w is a write, they both access the same location, and the value written by w is read by r. Unlike event structures, however, executions do have the equal write (ew) relation. (Since an execution has no conflicting events, then ew would necessarily be empty.) Similarly, the modification order of executions (X.mo) is total on writes to the same location.

In order to define execution consistency, we need some additional definitions, which are taken straight from Lahav et al. [2017] and are presented in Fig. 10. Here, *synchronizes-with* is defined as in C11 referring to the reads-from relation rather than jf, which does not exist at the level of executions. The last three definitions (scb, $\mathsf{psc}_{\mathsf{base}}$, and $\mathsf{psc}_{\mathsf{F}}$) are quite technical and concern cases when an ordering between SC events is guaranteed to be preserved by the memory model. They are not important for our examples, and are only included here for completeness.

Next, we define when an execution is consistent. As mentioned, we take the RC11 consistency constraints [Lahav et al. 2017], except that we allow $(\mathsf{X.po} \cup \mathsf{X.rf})^+$-cycles. The reason for allowing cycles is that we do not rely on RC11 consistency to rule out OOTA behaviors, but rather on the construction of event structures. One further (minor) difference is that we model updates as single events, whereas Lahav et al. [2017] model them as two events (a read and a write) connected by a special read-modify-write relation. As a result, our atomicity condition is slightly different.

Definition 5. An execution X is *consistent*, denoted isCons(X), if the following hold:

(Total-RF) $\mathsf{codom}(\mathsf{X.rf}) = \mathsf{X.E} \cap \mathcal{R}$;
(Total-MO) $\mathsf{X.mo}^= = ((\mathsf{X.E} \cap \mathcal{W}) \times (\mathsf{X.E} \cap \mathcal{W}))|_{\mathsf{loc}}$;
(Coherence) $\mathsf{X.hb}_{\mathsf{C11}} ; \mathsf{X.eco}^?$ is irreflexive;
(Atomicity) $\mathsf{X.fr} ; \mathsf{X.mo}$ is irreflexive; and
(SC) $\mathsf{X.psc}_{\mathsf{base}} \cup \mathsf{X.psc}_{\mathsf{F}}$ is acyclic.

where eco and fr are defined as for event structures.

The consistency constraints require that (1) every read of the execution reads from some write; (2) the modification order, mo, is total over all writes to the same location; (3) the execution is coherent (i.e., the execution's restriction to accesses of only one location is SC); (4) atomic updates read from their immediate mo-predecessor; and (5) certain cycles going through SC atomics are disallowed. Among other cases, it ensures that (1) putting SC fences between every two accesses in each thread guarantees SC, and (2) if all accesses of a consistent execution are sequentially consistent, then the execution is SC.

We move on to define how an execution is extracted from an event structure. First, we have to select an appropriate set of events, A, from the event structure so that restricting the event structure to those events gives us an execution. We place several constraints: (1) all those events should be visible; (2) they should not conflict with one another; and (3) they should be *prefix-closed* with respect to hb: i.e., every hb-predecessor of an event in A should also be in A; and Formally,

$$\mathsf{GoodRestriction}(G) \triangleq \left\{ A \;\middle|\; A \subseteq \mathsf{vis}(G) \;\wedge\; [A]\,;G.\mathsf{cf}\,;[A] = \emptyset \;\wedge\; \mathsf{dom}(G.\mathsf{hb}\,;[A]) \subseteq A \right\}$$

Second, given such a conflict-free set of events, $A \subseteq G.\mathsf{E}$, we define how to restrict G to get an execution. The following formal definition, $\mathsf{Project}_{\text{WEAKESTMO}}(G, A)$, does this by simply restricting po, rf, and mo to the set of events, A. $\mathsf{Project}_{\text{WEAKEST}}(G, A)$ is defined in a similar way. Since, however, WEAKEST does not record the modification order in event structures, $\mathsf{Project}_{\text{WEAKEST}}(G, A)$ generates an arbitrary mo.

$$\mathsf{Project}_{\text{WEAKESTMO}}(G, A) \triangleq \{\langle A\,,\, G.\mathsf{po} \cap (A \times A)\,,\, G.\mathsf{rf} \cap (A \times A)\,,\, G.\mathsf{mo} \cap (A \times A)\rangle\}$$

$$\mathsf{Project}_{\text{WEAKEST}}(G, A) \triangleq \{\langle A\,,\, G.\mathsf{po} \cap (A \times A)\,,\, G.\mathsf{rf} \cap (A \times A)\,,\, mo\rangle \mid mo \subseteq (A \times A)\}$$

Putting these two components together, the set of executions of event structure G are those consistent executions that are a result of projecting G to some good restriction of its events.

$$\mathsf{ex}_M(G) \triangleq \{\mathsf{X} \mid \exists A \in \mathsf{GoodRestriction}(G).\ \mathsf{X} \in \mathsf{Project}_M(G, A) \wedge \mathsf{isCons}_M(\mathsf{X})\}$$

Note that the executions extracted from G are not necessarily *maximal*. Rather, for any execution extracted from G, all its consistent (po ∪ rf)-prefixes can also be extracted from G.

3.4 Program Behaviors

The final step is to define program behaviors. We define the behavior of an execution to be the final contents of the memory at the end of an execution, i.e., the value written by the mo-maximal write for each location.

$$\mathsf{Behavior}(\mathsf{X}) \triangleq \{(e.\mathsf{loc}, e.\mathsf{wval}) \mid \exists e \in \mathsf{X.E} \cap \mathcal{W}.\ [\{e\}]\,;\mathsf{X.mo} = \emptyset\}$$

Then, the set of behaviors of a program under a model $M \in \{\text{WEAKEST}, \text{WEAKESTMO}\}$ contains the behaviors of any *maximal* execution X that can be extracted from an event structure G that was constructed from the program by following the model.

$$\mathsf{Behavior}_M(\mathbb{P}) \triangleq \{\mathsf{Behavior}(\mathsf{X}) \mid \exists G.\ \ G_{\mathsf{init}} \rightarrow_{\mathbb{P},M}{}^{*}\ G \;\wedge\; \mathsf{X} \in \mathsf{ex}_M(G) \;\wedge\; \mathsf{maximal}_{\mathbb{P}}(\mathsf{X})\}$$

where $\mathsf{maximal}_{\mathbb{P}}(\mathsf{X}) \triangleq \nexists i, lab.\ \mathsf{labels}(\mathsf{sequence}_{\mathsf{X.po}}(\mathsf{X.E}_i)) \cdot lab \in \mathbb{P}(i)$. Maximality ensures that all threads have terminated according to the thread semantics.

3.5 Java Causality Tests

We evaluate WEAKEST and WEAKESTMO on the Java causality tests [Manson et al. 2004]. Each test consists of a program along with a particular behavior and a remark as to whether that behavior should be allowed or not. For each test, we check whether our models can yield the behavior in question.

Our two models agree with each other and with PS on all the tests and with the prescribed Java behavior on 17/20 tests. Among the remaining cases, the behavior of test 16 is a classic coherence violation: while Java allows this outcome, our models forbid it. The other two cases (tests 19 and 20) are due to thread sequentialization (i.e., $\mathbb{C}_1 \,||\, \mathbb{C}_2 \rightsquigarrow \mathbb{C}_1 \,;\, \mathbb{C}_2$): Java allows the weak outcomes, whereas our models do not.

As a side note, we remark that the surpringly weak outcomes of tests 5 and 10 (forbidden by our models) can also be explained by thread sequentialization, yet Java forbids them.

4 THE WEAKEST MODEL AND PROMISING SEMANTICS

In this section, we relate the WEAKEST model to the promising semantics (PS) of Kang et al. [2017]. We first introduce PS and then, in §4.2, we show that WEAKEST is weaker than PS.

4.1 Overview of Promising Semantics

As already discussed in §2.4, PS is defined by an abstract machine. Its state $\mathsf{MS} = (\mathcal{TS}, \mathcal{S}, \mathsf{M})$ is a tuple with three components: $\mathcal{TS}$ is the *thread state map*, a function from thread identifiers to thread local states; $\mathcal{S}$ is the *global SC view*, which places a total order on the SC fences; and M is the *shared memory*, represented as a set of messages.

The thread state of thread i, $\mathcal{TS}(i)$, is again a tuple $TS = \langle \sigma, V, \mathsf{P} \rangle$ with three components. The first component, σ, stores the local state of the thread (e.g., the value of the program counter). As our models are parametric with respect to the syntax of the programming language, we take σ to be a pair of the set of traces of the thread, $\mathbb{P}(i)$, together with the sequence of labels produced so far. The second component, V, is the *thread view*, a mapping from memory locations to the maximum timestamp of a message that the thread is aware of.[2] The third component, P, records the set of outstanding promises made by thread i; i.e., the set of messages that thread i has added to the shared memory without yet having performed the corresponding writes.

As already mentioned, PS's memory, M, is just a set of messages, representing the writes (and promises) that have been performed. Each message m records the location written ($m.\mathsf{loc}$), the value written ($m.\mathsf{wval}$), a timestamp ($m.\mathsf{ts}$), as well as some other components that we will not use in this section.[3] For a location x, we write $\mathsf{maxmsg}(\mathsf{M}, x)$ for message in M with location x that has the maximum timestamp.

The initial thread state map, $\mathcal{TS}_{\mathsf{init}}(\mathbb{P})$, maps each thread i to the state $\langle \mathbb{P}(i), \epsilon \rangle$, the bottom view (that maps every location to the bottom timestamp), and the empty set of promises. The initial program state, $\mathsf{MS}_{\mathsf{init}}(\mathbb{P})$, has as components the initial thread state map, $\mathcal{TS}_{\mathsf{init}}(\mathbb{P})$, the bottom global SC view, and the initial memory, $\mathsf{M}_{\mathsf{init}}$, that contains an initialization message $\langle x : 0@\, \bot \rangle$ for each location x.

PS supports all the operations of our programming language except for SC accesses (reads, writes, and updates). It does support SC fences. Executing an SC fence simply updates the $\mathcal{S}$ component: it computes the elementwise maximal of the global SC view ($\mathcal{S}$) and its thread-local view (V), and updates both components to be that maximal view. In other words, for each location x, it sets both $\mathcal{S}$ and V to the maximum of $\mathcal{S}(x)$ and $V(x)$.

We have already informally discussed in §2.4 how the components of a state are affected by read, write, and promise transitions. For complete descriptions of these transitions and the formal definition of the promise machine, we refer the reader to Kang et al. [2017].

[2] Actually, each thread has three thread views: the *current* view, the *acquire* view, and the *release* view. The release and acquire views are used to assign appropriate semantics to release and acquire fences. Further details can be found in Kang et al. [2017] and in our technical appendix. Here, for simplicity, we describe only the current view.

[3] The additional components are another timestamp $m.\mathsf{ts}_{\mathsf{from}}$, useful for reserving timestamp ranges and giving appropriate semantics to updates, and a message view.

We take the behavior of a machine state in a promise execution to be its final memory contents excluding any unfulfilled promised messages. That is, for each location x, we select the non-promised message in M that has the maximal timestamp and return its value.

$$\mathsf{Behavior}(\mathsf{MS}) \triangleq \{(x, v) \mid x \in \mathsf{Locs} \wedge \mathsf{maxmsg}(\mathsf{MS.M} \setminus \bigcup_i \mathsf{MS}.\mathcal{TS}(i).\mathsf{P}, x).\mathsf{wval} = v\}$$

The behavior of a program $\mathbb{P}$ is the set of behaviors of all *final* machine states that can arise by executing the program (i.e., all machine states reachable from the initial program state that cannot reduce further):

$$\mathsf{Behavior}_{\mathsf{PS}}(\mathbb{P}) \triangleq \{\mathsf{Behavior}(\mathsf{MS}) \mid \mathsf{MS}_{\mathsf{init}}(\mathbb{P}) \rightarrow^* \mathsf{MS} \wedge \mathsf{MS} \not\rightarrow \}\,.$$

We note that recording the final memory contents suffices for distinguishing programs that differ only in their final thread-local states. We can place these programs in a context that writes the contents of the local states to main memory, and then use the definition above to distinguish them.

4.2 Relating the WEAKEST Model to Promising Semantics

We now state the main result of this section, which says that WEAKEST admits all the program behaviors that PS allows.

Theorem 1. *For a program* $\mathbb{P}$, $\mathsf{Behavior}_{\mathsf{PS}}(\mathbb{P}) \subseteq \mathsf{Behavior}_{\mathit{WEAKEST}}(\mathbb{P})$.

To prove this theorem, consider a final promise machine state MS of the program $\mathbb{P}$. We have to show that there exists a WEAKEST event structure G of $\mathbb{P}$ with a consistent execution $\mathsf{X} \in \mathsf{ex}_{\mathsf{WEAKEST}}(G)$ such that MS and X have same behavior. Our proof consists of three stages:

(1) First, we *define a simulation relation* $G \sim_\Pi \mathsf{MS}$ between the WEAKEST event structure G and the promise machine state MS. The relation is parameterized by a set of threads, Π, in which the event structure is lagging behind MS. The simulation relation relates various components of WEAKEST event structure to certain components of promise machine state by a number of mapping functions. We also define a set of relations to identify the desired execution in the WEAKEST event structure.

(2) Next, we prove that steps of the promising machine *preserve* the simulation relation. More precisely, starting from related states, every step of PS can be matched by zero or more steps of the WEAKEST event structure construction yielding related states. To handle promise steps, we split this proof obligation in two lemmas. Lemma 1 considers the transitions by non-promise operations of a particular thread, while Lemma 2 takes care of state transition on entire promise machine state.

(3) Finally, the simulation relation is defined so as to also identify an execution X of G such that the behavior of X and MS coincide. So, when PS reaches a final state, we extract that execution from G and complete the proof.

We move on to the definition of the simulation relation, $G \sim_\Pi \mathsf{MS}$, which says that G simulates MS modulo the threads Π. When $\Pi = \emptyset$, we write $G \sim \mathsf{MS}$ and say that G simulates MS. Our definition uses the following mapping functions, predicates, and relations.

- $\mathbb{W} : (G.\mathsf{E} \cap \mathcal{W}) \rightharpoonup \mathsf{M}$ is a partial function that maps certain write events of G to the corresponding messages in the promise machine memory.
- $\mathbb{S} \subseteq G.\mathsf{E}$ records the set of *covered events* in G, namely the events that are fully executed by the promising machine. In particular, any promised write in $\mathbb{S}$ must have been fulfilled in MS. We write $\mathbb{S}_i$ for the covered events of thread i (i.e., $\mathbb{S} \cap G.\mathsf{E}_i$).
- $\mathsf{sc} \subseteq (\mathbb{S} \cap \mathcal{F}_{\mathsf{sc}}) \times (\mathbb{S} \cap \mathcal{F}_{\mathsf{sc}})$ is a relation that enforces a total order on the sc fences following the $\mathcal{S}$ view of the promise machine. Note that PS disallows sc fence steps to appear in certification runs, which is why $\mathsf{sc} \subseteq \mathbb{S} \times \mathbb{S}$.

$$\mathsf{spo} \triangleq G.\mathsf{po} \cap (\mathbb{S} \times \mathbb{S}) \qquad \mathsf{srf} \triangleq G.\mathsf{rf} \cap (\mathbb{S} \times \mathbb{S}) \qquad \mathsf{smo} \triangleq \mathsf{mo} \cap (\mathbb{S} \times \mathbb{S})$$
$$\mathsf{sfr} \triangleq (\mathsf{srf}^{-1}; \mathsf{smo}) \setminus [G.\mathsf{E}] \qquad \mathsf{seco} \triangleq (\mathsf{srf} \cup \mathsf{smo} \cup \mathsf{sfr})^+$$
$$\mathsf{ssw} \triangleq [\mathcal{E}_{\sqsupseteq \mathrm{REL}}] ; ([\mathcal{F}] ; \mathsf{spo})^? ; [\mathcal{W}] ; \mathsf{spo}|_{\mathsf{loc}}^? ; [\mathcal{W}_{\sqsupseteq \mathrm{RLX}}] ; \mathsf{srf}^+ ; [\mathcal{R}_{\sqsupseteq \mathrm{RLX}}] ; (\mathsf{spo} ; [\mathcal{F}])^? ; [\mathcal{E}_{\sqsupseteq \mathrm{ACQ}}]$$
$$\mathsf{shb} \triangleq (\mathsf{spo} \cup \mathsf{ssw})^+$$

Fig. 11. Auxiliary definitions for the simulation relation.

Using these relations, we define $\mathsf{mo} \subseteq (G.\mathcal{W} \times G.\mathcal{W})|_{\mathsf{loc}}$ to order the writes to each location x according to the timestamps of the respective messages in the promise machine.

$$\mathsf{mo} \triangleq \{(e_1, e_2) \mid e_1.\mathsf{loc} = e_2.\mathsf{loc} \wedge \mathbb{W}(e_1).\mathsf{ts} < \mathbb{W}(e_2).\mathsf{ts}\} \setminus G.\mathsf{cf}$$

In Fig. 11, we further define restrictions of the program order, the reads-from relation, the modification order, the reads-before relation, the extended coherence order, the synchronization relation and the happens-before order to events in $\mathbb{S}$. Intuitively, the execution $\langle \mathbb{S}, \mathsf{spo}, \mathsf{srf}, \mathsf{smo} \rangle$ corresponds to the promise machine state.

We also define the behavior of the event structure with respect to $\mathbb{W}$ and $\mathbb{S}$ as follows.

$$\mathsf{Behavior}(G, \mathbb{W}, \mathbb{S}) \triangleq \{(x, v) \mid \exists e \in \mathcal{W}_x \cap \mathbb{S}.\ e.\mathsf{wval} = v \wedge \nexists e_1 \in \mathbb{S}.\ \mathsf{mo}(e, e_1)\}$$

Using these auxiliary definitions, we now define the simulation relation as follows.

Definition 6. Let $\mathbb{P}$ be a program with T threads, $\Pi \subseteq \mathsf{T}$ be a subset of threads, G be a WEAKEST event structure, and $\mathsf{MS} = \langle \mathcal{TS}, \mathcal{S}, \mathsf{M} \rangle$ be a promise machine state. We say that $G \sim_\Pi \mathsf{MS}$ holds iff there exist $\mathbb{W}$, $\mathbb{S}$, and sc such that the following conditions hold:

(1) G is consistent according to the WEAKEST model: $\mathsf{isCons}_{\mathrm{WEAKEST}}(G)$.
(2) The local state of each thread in MS contains the program of that thread along with the sequence of covered events of that thread: $\forall i.\ \mathcal{TS}(i).\sigma = \langle \mathbb{P}(i), \mathit{labels}(\mathsf{sequence}_{\mathsf{spo}}(\mathbb{S}_i)) \rangle$.
(3) Whenever $\mathbb{W}$ maps an event of G to a message in MS, then the location accessed and the written values match: $\forall e \in \mathsf{dom}(\mathbb{W}).\ e.\mathsf{loc} = \mathbb{W}(e).\mathsf{loc} \wedge e.\mathsf{wval} = \mathbb{W}(e).\mathsf{wval}$.
(4) All outstanding promises of threads $(\mathsf{T} \setminus \Pi)$ have corresponding write events in G that are po-after $\mathbb{S}$: $\forall i \in (\mathsf{T} \setminus \Pi).\ \forall e \in (\mathbb{S}_0 \cup \mathbb{S}_i).\ \mathcal{TS}(i).\mathsf{P} \subseteq \{\mathbb{W}(e') \mid (e, e') \in G.\mathsf{po}\}$.
(5) For every location x and thread i, the thread view of x in the promise state MS records the timestamp of the maximal write visible to the covered events of thread i.

$$\forall i, x.\ \mathcal{TS}(i).V(x) = \max\{\mathbb{W}(e).\mathsf{ts} \mid e \in \mathsf{dom}([\mathcal{W}_x]; G.\mathsf{jf}^?; \mathsf{shb}^?; \mathsf{sc}^?; \mathsf{shb}^?; [\mathbb{S}_i])\}$$

(6) The $\mathbb{S}$ events satisfy coherence: $\mathsf{shb}; \mathsf{seco}^?$ is irreflexive.
(7) The atomicity condition holds for the $\mathbb{S}$ events: $\mathsf{sfr}; \mathsf{smo}$ is irreflexive.
(8) The sc fences are appropriately ordered by sc: $[\mathcal{F}_{\mathrm{sc}}]; (\mathsf{shb} \cup \mathsf{shb}; \mathsf{seco}; \mathsf{shb}); [\mathcal{F}_{\mathrm{sc}}] \subseteq \mathsf{sc}$.
(9) The behavior of MS matches that of the $\mathbb{S}$ events: $\mathsf{Behavior}(\mathsf{MS}) = \mathsf{Behavior}(G, \mathbb{W}, \mathbb{S})$.

Next, we establish Lemmas 1 and 2, which concern the preservation of the simulation relation by program steps. First, we show that non-promising steps of threads i preserve simulation modulo thread i.

Lemma 1. $G \sim_{\{i\}} \mathsf{MS} \wedge \mathsf{MS} \xrightarrow{np}_i \mathsf{MS}' \implies \exists G'.\ G \to_{\mathbb{P}, \mathit{WEAKEST}}{}^* G' \wedge G' \sim_{\{i\}} \mathsf{MS}'$.

We prove this lemma by case analysis over the promise machine state transition. We simply perform the corresponding event structure construction step and define updated $\mathbb{W}$, $\mathbb{S}$, and sc to show that $G' \sim_{\{i\}} \mathsf{MS}'$ holds.

Using this lemma, we establish the following stronger property:

Lemma 2. $G \sim \mathsf{MS} \wedge \mathsf{MS} \rightarrow \mathsf{MS}' \implies \exists G'.\, G \rightarrow_{\mathbb{P},\, \mathit{WEAKEST}}{}^{*}\, G' \wedge G' \sim \mathsf{MS}'$.

To prove Lemma 2, we first use Lemma 1 to establish $G_1 \sim_{\{i\}} \mathsf{MS}'$ for an appropriate G_1. Then, we consider the PS certification run of thread i, $\mathsf{MS}' \xrightarrow{np}_i{}^{*} \mathsf{MS}''$ with $\mathsf{MS}''.\mathcal{TS}(i).\mathsf{P} = \emptyset$. We inductively apply Lemma 1 on this certification run to extend G_1 to G' such that $G' \sim_{\{i\}} \mathsf{MS}''$. Since, however, MS'' has no outstanding promises for thread i, it follows that $G' \sim \mathsf{MS}''$, and consequently also $G' \sim \mathsf{MS}'$ (for a smaller $\mathbb{S}$), as required.

Finally, Theorem 1 follows from Lemma 2 by induction.

5 DATA-RACE-FREEDOM GUARANTEES

In this section we define the notion of a data race and then show that our models provide certain guarantees for programs without races. These guarantees are stated as DRF ("data-race-freedom") theorems, and allow programmers to understand the behavior of a certain class of programs without needing to understand the WEAKESTMO model.

In the context of an execution X, we say that two non-conflicting events are *concurrent* if they are not related by the happens-before relation. Two concurrent events are *racy* if they access the same location and at least one of them is a write. Let $\mathsf{X}.\mathsf{Race}$ be the set of all racy events of execution X:

$$\mathsf{X}.\mathsf{Race} \triangleq \mathsf{dom}(((\mathsf{X}.\mathsf{E} \times \mathsf{X}.\mathsf{E}) \setminus (\mathsf{X}.\mathsf{hb}^{=}_{\mathsf{C11}} \cup \mathsf{X}.\mathsf{cf}))|_{\mathsf{loc}} \cap \mathsf{one}(\mathcal{W}))$$

where $\mathsf{one}(A)$ is the relation saying that at least one of its components belongs to the set A; that is, $\mathsf{one}(A)(x, y) \triangleq (x \in A \vee y \in A)$. An execution X is *RA-race-free* if its races are confined to SC accesses (i.e., $\mathsf{X}.\mathsf{Race} \subseteq \mathcal{E}_{\mathsf{sc}}$). Execution X is *RLX-race-free* if $\mathsf{X}.\mathsf{Race} \subseteq \mathsf{Ld}_{\sqsupseteq \mathsf{ACQ}} \cup \mathsf{St}_{\sqsupseteq \mathsf{REL}} \cup \mathsf{U}_{\sqsupseteq \mathsf{ACQ\text{-}REL}}$.

We recall the notion of RC11-consistency. An execution, X is RC11-consistent if it is consistent according to Definition 5 and $\mathsf{X}.\mathsf{po} \cup \mathsf{X}.\mathsf{rf}$ is acyclic. Then, we can establish the following result.

Theorem 2 (DRF-RLX). *Given a program $\mathbb{P}$, suppose its RC11-consistent executions are RLX-race-free. Then,* $\mathsf{Behavior}_{\mathit{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{\mathrm{RC11}}(\mathbb{P})$.

To prove this theorem, we first show that $G.\mathsf{jf} \subseteq G.\mathsf{hb}$ holds for every event structure G constructed from $\mathbb{P}$. It then follows that at most one full execution X can be extracted from G (with $\mathsf{X}.\mathsf{rf} = G.\mathsf{jf}$), and so that execution has $\mathsf{X}.\mathsf{po} \cup \mathsf{X}.\mathsf{rf}$ be acyclic. A more detailed proof can be found in our technical appendix [Chakraborty and Vafeiadis 2018].

Composing our DRF-RLX theorem with the DRF-SC theorem of Lahav et al. [2017, Theorem 4], we can derive a standard DRF-SC theorem, which says that programs whose races under SC are restricted to SC accesses exhibit only SC behavior.

Theorem 3 (DRF-SC). *Given a program $\mathbb{P}$, suppose its SC-consistent executions are RA-race-free. Then,* $\mathsf{Behavior}_{\mathit{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{\mathrm{SC}}(\mathbb{P})$.

We can further combine our DRF-RLX theorem with Lahav et al. [2017, Theorem 5] to get another simple criterion for ensuring the absence of weak behaviors. Namely, if all accesses of a program $\mathbb{P}$ to shared locations are atomic (i.e., at least RLX) and shared-location accesses in the same thread are separated by an SC-fence, then that program has only SC behaviors.

Finally, composing our DRF-RLX theorem with the DRF-RA theorem of Kang et al. [2017, Theorem 2], we can derive a DRF-RA theorem for WEAKESTMO. The DRF-RA theorem states that a program whose races under release-acquire consistency (i.e., by treating all accesses as having release/acquire semantics) are confined to release/acquire accesses (or stronger) exhibits only release-acquire consistent behavior.

Theorem 4 (DRF-RA). *Given a program $\mathbb{P}$, suppose its RA-consistent executions are RLX-race-free. Then,* $\mathsf{Behavior}_{\mathit{WEAKESTMO}}(\mathbb{P}) = \mathsf{Behavior}_{\mathrm{RA}}(\mathbb{P})$.

6 COMPILATION RESULTS

WEAKESTMO can serve as an intermediate memory model for an optimizing compiler from C/C++ to target architectures such as x86, PowerPC, and ARM. Typically, compilation takes place in multiple steps. First, the C/C++ program is mapped to the compiler's intermediate representation (IR), which will follow the WEAKESTMO semantics. Then, a number of optimizing transformations are performed in the IR, and finally the IR is mapped to the code for the respective target architecture. We first define the transformation correctness and then study the correctness of these transformation steps. The correctness proofs of these transformations are in our technical appendix [Chakraborty and Vafeiadis 2018].

Definition 7. A transformation of program $\mathbb{P}_{\text{src}}$ in memory model M_{src} to program $\mathbb{P}_{\text{tgt}}$ in model M_{tgt} is *correct* if it does not introduce new behaviors: i.e., $\text{Behavior}_{M_{\text{tgt}}}(\mathbb{P}_{\text{tgt}}) \subseteq \text{Behavior}_{M_{\text{src}}}(\mathbb{P}_{\text{src}})$.

Variants of WEAKESTMO. Concerning the semantics of data races, we define two variants of the WEAKESTMO model.

WEAKESTMO-C11 According to the C and C++ standards, if a program has a consistent execution with NA-race, then the program has *undefined* behavior: i.e., the program may generate any arbitrary outcome [ISO/IEC 14882 2011; ISO/IEC 9899 2011]. To model this semantics of races, we say that a program has arbitrary behavior if it contains an extracted execution with a NA-race.

WEAKESTMO-LLVM The LLVM concurrency semantics differs from C11 in handling racy programs [Chakraborty and Vafeiadis 2017]. LLVM distinguishes between *write-write* races and *read-write* races. A write-write race is one happening between a pair of write events, whereas a read-write race is between a load event and a concurrent write event.
According to LLVM, only write-write NA-races result in undefined behavior. In contrast, read-write NA-races have defined behavior: the racy read may return an **undef**(**u**) expression which can be materialized to an arbitrary constant value of the same data type.

The WEAKESTMO-C11 has the same set of rules as WEAKESTMO; the WEAKESTMO-LLVM construction rules differ and are shown in our technical appendix [Chakraborty and Vafeiadis 2018]. In the following discussion, 'WEAKESTMO' refers to both variants of the model.

6.1 Mapping from C/C++ to WEAKESTMO

WEAKESTMO supports all the features of the C11 model except for `memory_order_consume` loads. Thus, after strengthening the consume loads into acquire loads (a transformation that is meant to be sound in C11), the mapping from C11 to WEAKESTMO is simply the identity mapping.

We take the C11 concurrency model to be its revised formal definition, namely the weakRC11 model by Lahav et al. [2017]. Since we use the same weakRC11 model for constraining inconsistent executions (§3.3), the correctness of the mapping is straightforward.

Theorem 5. *The identity mapping from* weakRC11 *to WEAKESTMO is correct (for both models).*

6.2 Optimizations as WEAKESTMO Source-to-Source Transformations

We move on to consider the correctness of standard compiler optimizations. Following the related work [Vafeiadis et al. 2015; Ševčík 2011], we consider compiler optimizations as being composed of a number of simple thread-local source-to-source transformations—introductions, reorderings, and eliminations of memory accesses—and we restrict attention to the correctness of those basic transformations.

To describe these transformations on a thread i, we use the informal syntax $\alpha \rightsquigarrow \beta$, where α and β are sequences of memory accesses possibly with metavariables ranging over both α and β.

Table 1. Allowed reorderings $a{\cdot}b \rightsquigarrow b{\cdot}a$ where $\ell \neq \ell'$.

$\downarrow a \setminus b \rightarrow$	$\mathsf{Ld}_{\sqsubseteq \mathrm{ACQ}}(\ell)$	$\mathsf{Ld}_{\mathrm{SC}}(\ell)$	$\mathsf{St}_{\mathrm{NA}}(\ell)$	$\mathsf{St}_{\mathrm{RLX}}(\ell)$	$\mathsf{St}_{\sqsupseteq \mathrm{REL}}(\ell)$	$\mathsf{U}_{\sqsubseteq \mathrm{ACQ}}(\ell)$	$\mathsf{U}_{\sqsupseteq \mathrm{REL}}(\ell)$	$\mathsf{F}_{\mathrm{ACQ}}$	$\mathsf{F}_{\mathrm{REL}}$	F_{SC}
$\mathsf{Ld}_{\mathrm{NA}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{Ld}_{\mathrm{RLX}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
$\mathsf{Ld}_{\sqsupseteq \mathrm{ACQ}}(\ell')$	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
$\mathsf{St}_{\sqsubseteq \mathrm{REL}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{St}_{\mathrm{SC}}(\ell')$	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗
$\mathsf{U}_{\sqsubseteq \mathrm{REL}}(\ell')$	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
$\mathsf{U}_{\sqsupseteq \mathrm{ACQ}}(\ell')$	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
$\mathsf{F}_{\mathrm{ACQ}}$	✗	✗	✗	✗	✗	✗	✗	=	✗	✗
$\mathsf{F}_{\mathrm{REL}}$	✓	✓	✓	✗	✓	✗	✓	✓	=	✗
F_{SC}	✗	✗	✗	✗	✗	✗	✗	✗	✗	=

$$\mathsf{St}_{o'}(x, v'){\cdot}\mathsf{St}_{o}(x, v) \rightsquigarrow \mathsf{St}_{o}(x, v) \qquad \text{(Overwritten write)}$$
$$\mathsf{St}_{o}(x, v){\cdot}\mathsf{Ld}_{o'}(x, v) \rightsquigarrow \mathsf{St}_{o}(x, v) \qquad \text{(Read after write)}$$
$$\mathsf{U}_{o}(x, v', v){\cdot}\mathsf{Ld}_{o'}(x, v) \rightsquigarrow \mathsf{U}_{o}(x, v', v) \qquad \text{(Read after update)}$$
$$\mathsf{Ld}_{o}(x, v){\cdot}\mathsf{Ld}_{o'}(x, v) \rightsquigarrow \mathsf{Ld}_{o}(x, v) \qquad \text{(Read after read)}$$
$$\mathsf{Ld}_{\mathrm{NA}}(x, v){\cdot}\mathsf{St}_{\mathrm{NA}}(x, v) \rightsquigarrow \epsilon \qquad \text{(Non-atomic read write)}$$
$$\mathsf{St}_{\mathrm{NA}}(x, v'){\cdot}\tau{\cdot}\mathsf{St}_{\mathrm{NA}}(x, v) \rightsquigarrow \tau{\cdot}\mathsf{St}_{\mathrm{NA}}(x, v) \qquad \text{(Non-adjacent overwritten write)}$$
$$\mathsf{St}_{\mathrm{NA}}(x, v){\cdot}\tau{\cdot}\mathsf{Ld}_{\mathrm{NA}}(x, v) \rightsquigarrow \mathsf{St}_{\mathrm{NA}}(x, v){\cdot}\tau \qquad \text{(Non-adjacent read after write)}$$

Fig. 12. Safe eliminations where $o' \sqsubseteq o$ and τ does not contain any x-accesses nor any release-acquire pairs. That is, for all τ_1, τ_2 such that $\tau = \tau_1{\cdot}\tau_2$, either τ_1 does not contain a release label or τ_2 does not contain an acquire label.

By this syntax, we mean that every trace of memory accesses by the target code $\mathbb{P}_{\mathrm{tgt}}(i)$ is also a trace of the source code of $\mathbb{P}_{\mathrm{src}}(i)$ where at most one subsequence α of the trace has been replaced with β. More formally, $\mathbb{P}_{\mathrm{tgt}}(i) \subseteq \mathbb{P}_{\mathrm{src}}(i) \cup \{\tau{\cdot}\beta{\cdot}\tau' \mid \tau{\cdot}\alpha{\cdot}\tau' \in \mathbb{P}_{\mathrm{src}}(i)\} \wedge \forall j \neq i.\ \mathbb{P}_{\mathrm{tgt}}(j) = \mathbb{P}_{\mathrm{src}}(j)$.

Reordering of Independent Accesses. Compilers often reorder a pair of independent instructions as part of instruction scheduling or in order to enable further optimizations. The safe (✓) and unsafe (✗) reorderings are listed in Table 1. We prove the correctness of the safe transformations by constructing for each target execution a source execution with the same behavior.

Theorem 6. *The safe reorderings in Table 1 are correct in both* WEAKESTMO *models.*

Counterexamples for the unsafe reorderings can be found in Vafeiadis et al. [2015]. As a side remark, GCC and LLVM do not exploit the full range of correct reorderings: they typically reorder only non-atomic accesses with respect to other accesses, but do not reorder pairs of atomic accesses.

Redundant Access Elimination. We enlist a number of correct elimination transformations in Fig. 12. The first four transformations remove an access because of an adjacent access to the same location. Compilers like GCC and LLVM perform such transformations only when the eliminated access is non-atomic (i.e., $o' = \mathrm{NA}$); the eliminations are, however, also correct even for atomic accesses as long as the memory order of the eliminated access is not stronger than that of the justifying access (i.e., $o' \sqsubseteq o$). Combining the adjacent access eliminations with a number of safe reordering steps enables us to also eliminate certain non-adjacent accesses. Next, the fifth

WEAKESTMO	via RC11 [Lahav et al. 2017]			SPower
	x86	PowerPC	ARMv7	
$\text{store}_{\sqsubseteq \text{RLX}}$	mov	st	st	st
$\text{store}_{\text{REL}}$	mov	lwsync; st	dmb; st	lwsync; st
store_{SC}	mov; mfence	hwsync; st	dmb; st	hwsync; st
load_{NA}	mov	ld	ld	ld
load_{RLX}	mov	ld; cmp; bc	ld; cmp; bc	ld
load_{ACQ}	mov	ld; cbisync	ld; cbisb	ld; lwsync
load_{SC}	mov	hwsync; ld; cbisync	dmb; ld; cbisb	hwsync; ld; lwsync
CAS_{RLX}	lock cmpxchg	U	U	U
CAS_{REL}	lock cmpxchg	lwsync; U	dmb; U	lwsync; U
CAS_{ACQ}	lock cmpxchg	U; cbisync	U; cbisb	U, lwsync
$\text{CAS}_{\text{ACQ-REL}}$	lock cmpxchg	lwsync; U; cbisync	dmb; U; cbisb	lwsync; U; lwsync
CAS_{SC}	lock cmpxchg	hwsync; U; cbisync	dmb; U; cbisb	hwsync; U; lwsync
$\text{F}_{\sqsubset \text{SC}}$	mfence	lwsync	dmb	lwsync
F_{SC}	mfence	hwsync	dmb	hwsync

PowerPC $U \triangleq L$: lwarx; cmpw; bne L'; stwcx.; bne L; L': cbisync $\triangleq$ cmp; bc; isync

ARMv7 $U \triangleq L$: ldrex; mov; teq L'; strexeq; teq L; L': cbisb $\triangleq$ cmp; bc; isb

Fig. 13. Compilation schemes to x86, PowerPC, and ARM.

transformation removes a non-atomic load-store pair corresponding to the assignment $x = x$ in a programming language. The last two transformations eliminate a non-atomic access that is redundant because of a non-atomic store to the same location.

Theorem 7. *The eliminations in Fig. 12 are correct in both* WEAKESTMO *models.*

Speculative Load Introduction. One subtle transformation that the LLVM compiler performs is speculative load introduction. This happens in cases such as the following

$$\textbf{if}\,(cond)\ \ a = X_o; \quad \rightsquigarrow \quad t = X_o; \textbf{if}\,(cond)\ \ a = t;$$

where a conditional shared memory load is hoisted outside the conditional by introducing a fresh temporary variable, which is used only when the condition holds. As the conditional move between registers can be performed without a conditional branch, LLVM's transformation avoids introducing a branch instruction at the cost of possibly performing an unnecessary load (if the condition is false).

This transformation is incorrect in the WEAKESTMO-C11 model because it may introduce a data race in the program when *cond* is false. It is, however, correct in WEAKESTMO-LLVM model.

Theorem 8. *The transformation* $\epsilon \rightsquigarrow \text{Ld}_o(x, \_)$ *is correct in* WEAKESTMO-LLVM.

Strengthening. Finally, it is sound to strengthen the memory order of an access or fence (e.g., $\text{F}_o \rightsquigarrow \text{F}_{o'}$ for $o \sqsubset o'$) as well as to introduce a new fence instruction in a program (i.e., $\epsilon \rightsquigarrow \text{F}_o$). These transformations are correct because WEAKESTMO is monotone.

6.3 Mapping from WEAKESTMO to x86, PowerPC, and ARMv7

Finally, we present some results about the lowering transformations from WEAKESTMO to the x86, PowerPC, and ARMv7 architectures.

First, by observing that WEAKESTMO is weaker than RC11, we use the RC11 mapping correctness results of Lahav et al. [2017] to get correct mappings to the aforementioned architectures (see Fig. 13). While these mappings are correct, they are suboptimal for PowerPC and ARMv7 in that they insert a fake conditional branch after each relaxed load.

To resolve this problem, we also prove the correctness of the intended mapping to the SPower model, a stronger version of the Power model due to Lahav and Vafeiadis [2016]. In addition to the Power model's constraints, SPower requires $(po \cup rf)$ to be acyclic, which makes the fake control dependencies after relaxed loads unnecessary. The interest of the compilation scheme to SPower is twofold. First, even though load-store reordering is architecturally allowed by the PowerPC model, as far as we know, no existing implementations actually perform such reorderings, and hence SPower is a correct description of existing implementations (at the time of writing). Second, Lahav and Vafeiadis [2016] show that excluding CAS and SC accesses, PowerPC is equivalent to a model that first transforms the program by arbitrarily reordering independent memory accesses and then returns examines the behaviors of the transformed program according the SPower model. Thus, following the proof strategy of Kang et al. [2017], since WEAKESTMO supports reorderings of independent memory accesses, to prove correctness of compilation to PowerPC, it suffices to prove correctness of compilation to SPower.

Proving the correctness of the intended optimal mappings from WEAKESTMO to the full PowerPC model and to ARMv7, as well as to the ARMv8-Flat model [Pulte et al. 2018], is left for future work. To carry out these proofs, we intend to compile via the recent IMM model of Podkopaev et al. [2019], from which compilation results to multiple architectures have been developed.

7 RELATED WORK

As explained in the introduction, defining an adequate concurrency model that does not allow *out-of-thin-air* behaviors has been a longstanding research challenge, and has led to a number of proposals. Among them, we are aware of five that correctly differentiate CYC, LB, and LBfd.

In an early attempt, Manson et al. [2005] defined a fairly complicated operational model for Java. Their model, however, turned out to be unsound: Ševčík and Aspinall [2008] showed that it disallows some outcomes that were actually observable on Java programs. Nevertheless, they provided a set of 20 causality benchmarks, which became a standard benchmark for comparing memory models. (We discuss the outcomes of our models on those benchmarks in §3.5.)

Jeffrey and Riely [2016] later defined a model for a fragment of Java based on event structures. Similar to the Java memory model, executions are constructed by adding one event at a time. Each newly added read event has to be AE-justified: for all moves of the opponent (choosing how to continue executing the program), there must exist an extension containing a write whence the read gets its value. While AE-justification assigns the right semantics to the RNG example, its additional complexity over our more basic existential justification leads to some problems. First, enumerating all possible executions of a program is computationally infeasible, because one has to prove an assertion with N quantifier alternations (where N is the length of the execution). Second, their model fails to validate the reordering of independent read events, and therefore cannot be compiled to Power and ARM without additional fences. The paper, however, sketches an extension of the model that seems to fix this problem. It remains unclear whether the extension satisfies DRF-SC and can be compiled to hardware with the intended mappings.

Pichon-Pharabod and Sewell [2016] proposed a model for a fragment of C11 based on plain event structures (with only program order and conflict edges). They define an operational semantics over such event structures that iteratively constructs an execution. It does so by either committing a minimal event from the structure or by performing one of set of pre-determined compiler optimizations that rewrites the remaining event structure. The resulting model is quire complicated

and, as such, does not have much metatheory developed about it. It also fails to explain some weak program behaviors (cf. the ARM-weak program of [Kang et al. 2017]).

Chakraborty and Vafeiadis [2017] developed another model based on event structures for a fragment of LLVM concurrency without relaxed accesses (a.k.a. monotonic in LLVM's terminology). As such, their event structures are substantially simpler than the ones in this paper as, e.g., the distinction between WEAKEST and WEAKESTMO does not arise in their restricted setting. This paper can be seen as an extension of that work both in terms of C11 feature coverage and also of the results proved.

Kang et al. [2017] defined the promising semantics, which comes with a number of results, which we are able to exploit for our WEAKEST model. It has also led to some follow up work, which proved correctness of compilation to ARM [Podkopaev et al. 2017, 2019] and the correctness of a program logic over it [Svendsen et al. 2018]. Nevertheless, as discussed, the promising semantics also has a number of shortcomings, which are difficult to resolve because of the model's complexity and brittleness. We believe there are three main reasons for this complexity: (1) the use of timestamps and messages to represent executions as opposed to execution graphs; (2) the ability to promise a write at any point during execution in conjunction with the certification of promises, which can be of arbitrary length, which make it difficult to use the promising semantics as an execution oracle; and (3) the quantification over all 'future' memories in the certification of promises [Kang et al. 2017, §3], which is needed for handling RMWs. Its shortcomings include (a) admitting some questionable behaviors (e.g., that of Coh-CYC); (b) not supporting global optimizations; (c) providing overly strong semantics of RMWs, whose compilation to ARMv8 incurs an unintended performance cost (cf. FADD); and (d) not supporting SC accesses. Our work provides solutions to problems (a), (c), and (d), albeit our solution for (c) is not supported by a compilation correctness proof.

Besides these proposals, some papers have argued for disallowing the weak outcome of LB at the expense of requiring a more expensive compilation scheme to Power and ARM [Batty et al. 2013; Boehm and Demsky 2014; Lahav et al. 2017; Ou and Demsky 2018; Vafeiadis and Narayan 2013]. This seems adequate for the C/C++ setting, which forbids races on non-atomic accesses. It is, however, considered to be too expensive for Java, where for type-safety reasons all races must have defined semantics. Our work actually shows that one can still provide a sound and useful model that does allow the weak outcomes of LB and LBfd.

8 FUTURE WORK AND CONCLUSION

In this paper, we proposed a memory model based on event structures. Our model has two variants: the weaker version, WEAKEST, which simulates the promising semantics [Kang et al. 2017] and the stronger version, WEAKESTMO, which removes certain questionable behaviors allowed in promising semantics. We showed that the WEAKESTMO resolves out-of-thin-air problem, provides standard DRF guarantees, allows the desirable optimizations, and can be mapped to the architectures like X86, PowerPC, ARMv7. Additionally our models are flexible enough to adopt existing results from other papers [Chakraborty and Vafeiadis 2017; Lahav and Vafeiadis 2016; Lahav et al. 2017].

Even though WEAKESTMO and promising semantics are not comparable, we observe the same outcomes for the examples in Kang et al. [2017]. Specifically, similar to promising semantics, we allow the weak outcome of the program in Kang et al. [2017, §6]. Also, in the example in Kang et al. [2017, §7], WEAKESTMO disallows $x = 1$ in the source but allows the $x = 1$ outcome in the target program after sequentialization. Thus, similar to promising semantics, WEAKESTMO cannot handle sequentialization.

In the future, we intend to prove the correctness of the mappings from WEAKESTMO to the new ARMv8 model [Pulte et al. 2018] as well as the optimal mappings to PowerPC model [Alglave et al.

2014] via IMM [Podkopaev et al. 2019]. Additionally, we would like to mechanize our results in an interactive proof assistant, so as to provide further confidence about their correctness.

Further ahead, for WEAKESTMO to be established as a good programming language consistency model, we would have to develop techniques to reason about concurrent programs running in the WEAKESTMO model.

In terms of *program logics*, there are several works handling different fragments of RC11 (e.g., [Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013]). Their soundness proofs, however, rely heavily on the acyclicity of po ∪ rf and it is unclear whether they can be adapted to the weaker setting of WEAKESTMO. Among these logics, RSL [Vafeiadis and Narayan 2013] should be sound under our model, because Svendsen et al. [2018] proved the soundness of a variant of it over the promising semantics. For the more advanced logics, however, the counterexample of Doko and Vafeiadis [2017] shows that certain adaptations will be needed.

In terms of *model checking* for weak memory concurrency, again there are a number of works that assume po ∪ rf acyclicity (e.g., [Abdulla et al. 2017; Kokologiannakis et al. 2018]). There are also tools that do not require po ∪ rf acyclicity [Abdulla et al. 2016; Alglave et al. 2014; Norris and Demsky 2016]. These tools, however, use more expensive state enumeration techniques, and are significantly slower than the tools that assume po ∪ rf acyclicity. It remains to be seen whether similar model checking approaches can work for models based on event structures, and if so, how much slower they would be in comparison to the existing tools.

ACKNOWLEDGMENTS

We would like to thank Marko Doko, Evgenii Moiseenko, Anton Podkopaev, Azalea Raad, and the anonymous POPL reviewers for their feedback on earlier drafts of this paper.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless model checking for TSO and PSO. *Acta Inf.* 54, 8 (2017), 789–818. https://doi.org/10.1007/s00236-016-0275-0

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless model checking for POWER. In *CAV 2016 (LNCS)*, Vol. 9780. Springer, Heidelberg, 134–156. https://doi.org/10.1007/978-3-319-41540-6_8

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—A new definition. In *ISCA 1990*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *ASPLOS 2018*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *POPL'13*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2429069.2429099

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL'11*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

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

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *CGO 2017*. IEEE, Piscataway, NJ, USA, 100–110. https://doi.org/10.1109/CGO.2017.7863732

Soham Chakraborty and Viktor Vafeiadis. 2018. Technical appendix. Available at http://plv.mpi-sws.org/weakest/.

Marko Doko and Viktor Vafeiadis. 2016. A program logic for C11 memory fences. In *VMCAI 2016 (LNCS)*, Vol. 9583. Springer, Heidelberg, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP 2017 (LNCS)*, Vol. 10201. Springer, Heidelberg, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

ISO/IEC 14882. 2011. Programming Language C++.

ISO/IEC 9899. 2011. Programming Language C.

Alan Jeffrey and James Riely. 2016. On thin air reads: Towards an event structures model of relaxed memory. In *LICS 2016*. ACM, New York, NY, USA, 759–767. https://doi.org/10.1145/2933575.2934536

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 2017 (LIPIcs)*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

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

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

Ori Lahav and Viktor Vafeiadis. 2016. Explaining relaxed memory models with program transformations. In *FM 2016 (LNCS)*, Vol. 9995. Springer, Heidelberg, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

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

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Jeremy Manson, William Pugh, and Sarita V. Adve. 2004. Causality Test Cases. http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html.

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

Brian Norris and Brian Demsky. 2016. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51. https://doi.org/10.1145/2806886

Peizhao Ou and Brian Demsky. 2018. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 136:1–136:29. https://doi.org/10.1145/3276506

Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL'16*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2676726.2676995

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

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

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

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP 2018 (LNCS)*, Vol. 10801. Springer, Heidelberg, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak-memory with ghosts, protocols, and separation. In *OOPSLA 2014*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

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*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

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

Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI'11*. ACM, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

Jaroslav Ševčík and David Aspinall. 2008. On validity of program transformations in the Java memory model. In *ECOOP'08*. Springer, Heidelberg, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

Glynn Winskel. 1986. Event Structures. In *Advances in Petri Nets*. Springer, Heidelberg, 325–392. https://doi.org/10.1007/3-540-17906-2_31