Relaxed Memory Concurrency Re-executed

EVGENII MOISEENKO, JetBrains Research, Serbia
MATTEO MELUZZI, TU Delft, the Netherlands
INNOKENTII MELESHCHENKO, JetBrains Research, Neapolis University Pafos, Cyprus
IVAN KABASHNYI, JetBrains Research, Constructor University Bremen, Germany
ANTON PODKOPAEV, JetBrains Research, Constructor University Bremen, the Netherlands
SOHAM CHAKRABORTY, TU Delft, the Netherlands

Defining a formal model for concurrency in programming languages that addresses conflicting requirements from programmers, compilers, and architectures has been a long-standing research question. It is widely believed that traditional axiomatic per-execution models that reason about individual executions do not suffice to address these conflicting requirements. Consequently, several multi-execution models were proposed that reason about multiple executions together. Although multi-execution models were major breakthroughs in satisfying several desired properties, these models are complicated, challenging to adapt to existing language specifications given in per-execution style, and they are typically not friendly to automated reasoning tools.

In response, we propose a re-execution-based memory model (XMM). Debunking the beliefs around per-execution and multi-execution models, XMM is (almost) a per-execution model. XMM reasons about individual executions, but unlike traditional per-execution models, it relates executions by a re-execution principle. As such, the memory consistency axioms and the out-of-order re-execution mechanics are orthogonal in XMM, allowing to use it as a semantic framework parameterized by a given axiomatic memory model.

We instantiated the XMM framework for the RC20 language model, and proved that the resulting model XC20 provides DRF guarantees and allows standard hardware mappings and compiler optimizations. Noteworthy, XC20 is the first model of its kind that also supports thread sequentialization optimization. Moreover, XC20 is also amenable to automated reasoning. To demonstrate this, we developed a sound model checker XMC and evaluated it on several concurrency benchmarks.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Semantics**.

Additional Key Words and Phrases: Weak memory models, C/C++ memory model, model checking

ACM Reference Format:
Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. 2025. Relaxed Memory Concurrency Re-executed. *Proc. ACM Program. Lang.* 9, POPL, Article 72 (January 2025), 62 pages. https://doi.org/10.1145/3704908

1 Introduction

Shared *relaxed* memory concurrency is the de facto programming paradigm for modern multicore architectures and programming languages. Under relaxed memory models, programs exhibit

Authors' Contact Information: Evgenii Moiseenko, JetBrains Research, Serbia, evgeniy.moiseenko@jetbrains.com; Matteo Meluzzi, TU Delft, the Netherlands, m.meluzzi@student.tudelft.nl; Innokentii Meleshchenko, JetBrains Research, Neapolis University Pafos, Cyprus, ; Ivan Kabashnyi, JetBrains Research, Constructor University Bremen, Germany, ; Anton Podkopaev, JetBrains Research, Constructor University Bremen, the Netherlands, apodkopaev@constructor.university; Soham Chakraborty, TU Delft, the Netherlands, s.s.chakraborty@tudelft.nl.

ACM 2475-1421/2025/1-ART72
https://doi.org/10.1145/3704908

additional executions compared to the executions under interleaving semantics, formally known as sequential consistency (SC) [Lamport 1979].

By now, relaxed memory concurrency has emerged as a first-class primitive in several mainstream programming languages, including C/C++ and Java [ISO/IEC 14882 2011; ISO/IEC 9899 2011; Manson et al. 2005a]. However, defining a semantic model for relaxed memory concurrency in programming languages has been a nontrivial challenge as it requires meeting the conflicting requirements of programmability and performance. We explain this tension in the following example programs where X and Y are shared variables initialized to 0.

```
      X = Y = 0
a = X;        || b = Y;
if (a == 1)   || if (b == 1)
  Y = 1;      ||   X = 1;
```
(LBD)

```
   X = Y = 0
a = X; || b = Y;
Y = 1; || if (b == 1)
       ||   X = 1;
```
(LB)

```
      X = Y = 0
a = X;        ||
if (a == 1)   || b = Y;
  Y = 1;      || if (b == 1)
else          ||   X = 1;
  Y = 1;      ||
```
(LBfd)

Consider the program LBD, LB, and LBfd. The execution with the outcome $a = b = 1$ should be forbidden for the program LBD as no mainstream architecture exhibits this behavior, and no program transformation can result in a program that exhibits this behavior. On the other hand, the same outcome should be possible for the LB program. This outcome can arise if the accesses in the first thread are reordered, and then the second thread is executed between the accesses in the first thread. The outcome should also be possible for the LBfd program as a compiler may merge the control flow branches in the first thread to generate the LB program.

However, under C/C++ concurrency semantics, known as C11 [Batty et al. 2011], all these programs exhibit the outcome $a = b = 1$. In the case of the LBD, this outcome is an *out-of-thin-air* (OOTA) outcome. Thus, the C11 model does not provide the *data-race-freedom (DRF) guarantee* — the program LBD exhibits a relaxed (non-SC) outcome, even though the program is race-free.

The example above demonstrates the limitation of the C11 concurrency model as it fails to satisfy the desiderata of conflicting requirements to a programming language relaxed memory model.

- For programmability, the model must have strong enough constraints to achieve *data-race-freedom* guarantees for well-synchronized programs and to forbid infamous OOTA executions.
- For performance, the semantic model must be weak enough to achieve optimal mapping schemes to the underlying architectures and to allow the desired program transformations performed by optimizing compilers.
- In addition, the semantics preferably should be executable and support the development of automated reasoning tools.

As was demonstrated by Batty et al. [2013], ‘per-execution’ based semantic models such as C11 cannot satisfy these requirements. This is because the executions are analyzed in an isolated manner, and multiple programs may share the same execution. Consequently, the original C11 model proposed by Batty et al. [2011] was strengthened by [Lahav et al. 2017] under the name RC11 to achieve data race freedom guarantees at the cost of suboptimal compilation.

In contrast to the per-execution models, the *multi-execution models* analyze multiple executions together to differentiate the programs more precisely [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2020; Jeffrey and Riely 2016; Jeffrey et al. 2022a; Kang et al. 2017; Paviotti et al. 2020; Pichon-Pharabod and Sewell 2016]. For instance, the promising semantics (PS) model introduces special features such as ‘promise’ and ‘certificates’ to analyze alternative executions while operationally constructing an execution [Kang et al. 2017]. Event structure-based approaches [Chakraborty and Vafeiadis 2019; Jeffrey and Riely 2016; Paviotti et al. 2020; Pichon-Pharabod and Sewell 2016] capture multiple executions in a single event structure to differentiate between programs.

However, being defined in a very different style using complex abstractions, the multi-execution semantics are hard to integrate into the existing per-execution-based language specifications such as C/C++ or Java. Moreover, the multi-execution frameworks are tightly integrated with the consistency constraints of a memory model: defining another model with different constraints may require overhauling the entire semantics from the ground up. Finally, existing multi-execution models still struggle to support some program transformations; most notably, thread sequentialization has proven to be quite challenging.

Thus, defining a formal semantic model for modern relaxed concurrency that meets all the requirements have remained a long-standing problem. To address this problem, we propose XMM, an (almost) per-execution-based semantic framework that satisfies the desiderata of properties discussed above in a single model. We instantiate the framework on the C/C++ concurrency model, in particular, its C20 revision (that is RC20 modulo porf acyclicity) [Margalit and Lahav 2021], to obtain a memory model we call XC20.

To the best of our knowledge, XC20 is the **first model that supports *thread sequentialization* transformation** in combination with local reordering and elimination transformations, and the data-race-freedom guarantee. Sequentialization merges two threads into one, that is, $T_1 \mid\mid T_2 \rightsquigarrow T_1; T_2$. It is an important transformation for parallel programs, where each iteration of a parallel loop results in concurrently running threads. For such programs, loop unrolling transformation implies sequentializing multiple threads.

Main idea. XMM is an *axiomatic-operational* semantic framework that consists of two steps: (in-order) execution and (out-of-order) re-execution. By in-order execution, XMM can construct a consistent execution following the *program-order* and *read-from* relations (a.k.a. porf). Next, given a consistent execution, XMM may *commit* a subset of its events and perform a *re-execution* following a set of constraints and by preserving the set of committed events to generate another execution. The re-execution steps enable XMM to create executions with one or more cycles consisting of porf relations in an operational manner. Thus, XMM checks the consistency of each execution individually, similar to the per-execution models, yet is able to relate different executions. We note that the execution construction steps in XMM are general enough to parameterize XMM over the consistency constraints and thread-local operational semantic steps for reconstructing the committed events during re-execution.

XMM is verification-friendly as it is an executable semantic framework due to the operational steps. To this end, we have developed a model checker XMC. The model checker explores the C20 executions of a given program including the ones with porf cycles. To ensure the correctness of the model checker, we have proved its soundness. Our experimental evaluation shows that the model checker can effectively reason about the existing benchmarks for weak memory concurrency. It is a major step, as most of the state-of-the-art weak memory model checkers such as GenMC [Kokologiannakis and Vafeiadis 2021] or Nidhugg [Abdulla et al. 2017] construct executions following porf order and hence cannot reason about executions with porf cycles. The existing model checkers following the multi-execution models, to our knowledge, are either not proven sound (e.g. [Moiseenko et al. 2022]) or do not handle all primitives (e.g. [Pulte et al. 2019]).

The rest of the paper is structured as follows.

- §2 overviews the key aspects of XMM with examples.
- §3 provides formal semantics of XMM.
- §4 describes the XMC model checking algorithm and its soundness proof.
- §5 discusses the experimental results of XMC.

Finally, the additional technical details and proofs can be found in the supplementary material [Moiseenko et al. 2025].

2 Overview

This section provides a high-level explanation of the XMM semantics.

2.1 Per-Execution Axiomatic Models for Concurrency

In per-execution axiomatic models for concurrency, a program is represented by a set of *consistent finite execution graphs*. An execution graph consists of events as nodes resulting from shared memory accesses or fences, and various relations as edges among the events. The semantic models define a set of axioms using the events and relations to check if an execution is consistent.

Depending on the access type, an event is a read (R), write (W), or fence (F). Following C20, a read event can be non-atomic (NA) or atomic with relaxed (RLX) or acquire (ACQ) memory order. A write event can be non-atomic (NA), or atomic with relaxed (RLX) or release (REL) memory order. A fence event is atomic with acquire (ACQ), release (REL), or acquire-release (ACQ-REL) memory order. Considering the strength of the memory orders $\textsc{na} \sqsubseteq \textsc{rlx} \sqsubseteq \{\textsc{rel}, \textsc{acq}\} \sqsubseteq \textsc{acq-rel}$ holds. Unless mentioned, the events in the concurrently running threads have relaxed atomic memory order (RLX), and all memory locations are initialized to 0.

An execution contains the following primitive relations between the events.

- Relation program-order (po) denotes the syntactic order between the events.
- Relation reads-from (rf) relates a pair of write and read events with the same written and read values where the read event has read from the write event. Moreover, every read event reads from exactly one write event.
- Relation porf denotes a transitive closure of the union of the program order and reads-from.
- Relation modification-order (mo) is a strict total order over same-location writes.
- A successful read-modify-write (RMW) operation results in an rmw relation between a pair of read and write events on the same location which are also immediate-po related. A failed RMW results in only a single read (R) event.

For example, Figure 1 shows the execution graphs of the LBD, LB, and LBfd programs which are consistent under the C20 model.

Limitation. In the per-execution approach, multiple programs may share an execution graph, and the execution can be allowed or forbidden based on the consistency constraints. For example, the execution in Figure 1d is allowed by C20 but forbidden in RC20 [Margalit and Lahav 2021].

The RC20 model forbids this execution by requiring the acyclicity of the porf relation to achieve DRF guarantees. Consequently, RC20 not only forbids read-write reordering in compiler optimization but also results in suboptimal mapping of a read as it introduces a trailing fence for architectures such as ARM, Power, and RISC-V to forbid out-of-order read-write execution. On the other hand, C20 allows the execution, leading to an *out-of-thin-air* behavior in the LBD program. Thus, the traditional per-execution-based models suffer from a tradeoff between optimization and programmability guarantees.

Execution construction. The acyclicity of (po ∪ rf) relation in RC20 provides an in-order execution construction mechanism for several analyses and verification approaches [Abdulla et al. 2015; Doko and Vafeiadis 2016; Kokologiannakis and Vafeiadis 2021; Luo and Demsky 2021; Norris and Demsky 2013]. At each construction step, an existing execution graph is extended with a new porf maximal event, and then the consistency of the updated execution is checked. For example, the executions from Figures 1a to 1c can be constructed following the porf order, preserving the RC20 consistency constraints at each step. Note that the in-order execution construction cannot generate the execution graph shown on Figure 1d though LB and LBfd programs may exhibit this execution under the C20 model.

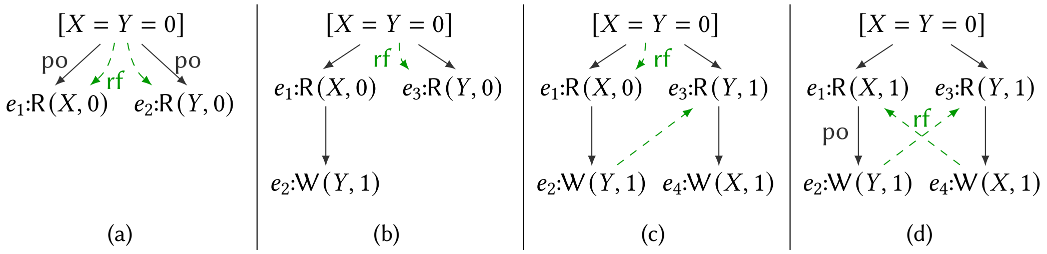


Fig. 1. C20 execution graphs. Figure 1a execution is allowed in LBD program but forbidden in LB and LBfd. The executions Figure 1b and Figure 1c are allowed for the LBfd and LB programs but are forbidden for the LBD program. Figure 1d is allowed in the LBD, LBfd, and LB programs.

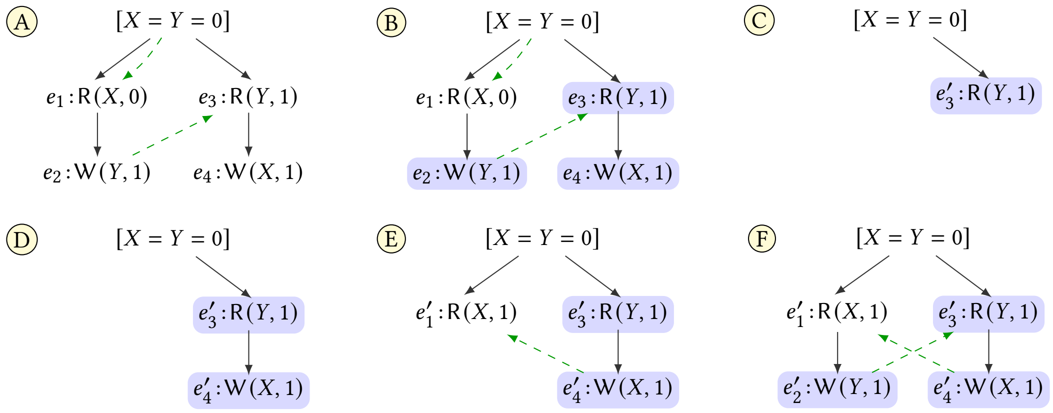


Fig. 2. XMM construction of the execution in Figure 1d with $(\mathsf{po} \cup \mathsf{rf})$ cycle for the LB and LBfd programs.

As such, models like RC20 avoid thin-air values by conservatively forbidding all porf cycles, ensuring that reads can only be added to an execution graph after their corresponding writes. On the other hand, if one drops porf acyclicity, naive graph construction can lead to thin-air reads, as shown by LBD example. To avoid thin-air reads and at the same time do not forbid porf cyclic graphs altogether we need some middle ground. The XMM semantics provide such an approach.

2.2 XMM: A Re-Execution based Axiomatic-Operational Model

XMM constructs execution graphs using two operational rules. The in-order execution rule works similarly as in porf acyclic models, by adding one porf maximal event at a time. The out-of-order re-execution rule starts with a given execution graph, selects a subset of its *commited events*, and re-builds the graph while preserving the committed subgraph. During the re-building process, adding a read that reads from "nowhere" is allowed only if it was committed in the original graph, ensuring that its value is grounded by the original execution. Ultimately, all such reads are restored to read from committed writes.

Let us consider an example in Figure 2, where we construct the execution graph from Figure 1d for the LB program (similar construction applies to the LBfd program).

- Ⓐ First, we construct the RC20 consistent graph from Figure 1c following the porf order.
- Ⓑ Given this execution, we commit events e_2, e_3, e_4, and re-execute the program in Ⓒ – Ⓕ.

- Ⓒ Next we append the event e_3' that preserves the same label as the committed event e_3 in Ⓑ. Note that the event e_3' does not read from any write and hence has no incoming rf edge.
- Ⓓ Further, we append a committed event e_4'.
- Ⓔ After this, we append the read event e_1' which now reads from an existing write event e_4'.
- Ⓕ Finally, we append the event e_2' that restores the committed event $\mathsf{W}(Y, 1)$ and creates the read from edge $\langle e_2', e_3' \rangle \in \mathsf{rf}$. The resulting execution graph is checked to be C20 consistent.

XMM model checking. The re-execution step does not provide any constraint on committing the subset of events in an execution. However, our model checker XMC judiciously commits the required events to construct the desired execution with porf cycle. XMC identifies so-called *load buffering races* [Moiseenko et al. 2022], that is read-write data races such that there exists a porf path from the read to the write event. We discuss the approach in detail in §4.

2.3 Sequentialization

The sequentialization transformation together with other transformations can enable porf cycles. Consider the following example from Kang et al. [2017] where a sequence of transformations following the sequentialization transformation results in the outcome $a = 1$ denoting an execution with a porf cycle.

$$\begin{array}{c} X = Y = 0 \\ \begin{array}{l} a = X; \\ \textbf{if}\,(a \neq 1) \\ \quad X = 1; \end{array} \left\| \; Y = X; \; \right\| X = Y; \end{array} \;\text{(SEQ)} \rightsquigarrow \begin{array}{c} X = Y = 0 \\ \begin{array}{l} a = X; \\ \textbf{if}\,(a \neq 1) \\ \quad X = 1; \\ Y = X; \end{array} \left\| \; X = Y; \right. \end{array} \;\text{(SEQ1)} \rightsquigarrow \begin{array}{c} X = Y = 0 \\ \begin{array}{l} a = X; \\ \textbf{if}\,(a \neq 1) \\ \quad X = 1; \\ Y = 1; \end{array} \left\| \; X = Y; \right. \end{array} \;\text{(SEQ2)} \rightsquigarrow \begin{array}{c} X = Y = 0 \\ \begin{array}{l} Y = 1; \\ a = X; \\ \textbf{if}\,(a \neq 1) \\ \quad X = 1; \end{array} \left\| \; X = Y; \right. \end{array} \;\text{(SEQ3)}$$

Firstly, the second thread is merged with the first thread. In SEQ2, following the conditional assignment of X in the first thread, the read of X always returns 1 in the last statement. Next, $Y = 1$ is reordered above in SEQ3 program. Finally, the SEQ3 program has an interleaving that executes the second thread between $Y = 1$ and $a = X$ in the first thread, leading to the outcome $a = 1$.

Supporting the sequentialization transformation with other desired properties has been a long-standing problem for the concurrency models. Sequentialization is sound in the RC20 model [Lahav et al. 2017], but this model forbids the outcome $a = 1$ for the source program nonetheless, because the corresponding execution graph contains a porf cycle. On the other hand, sequentialization is

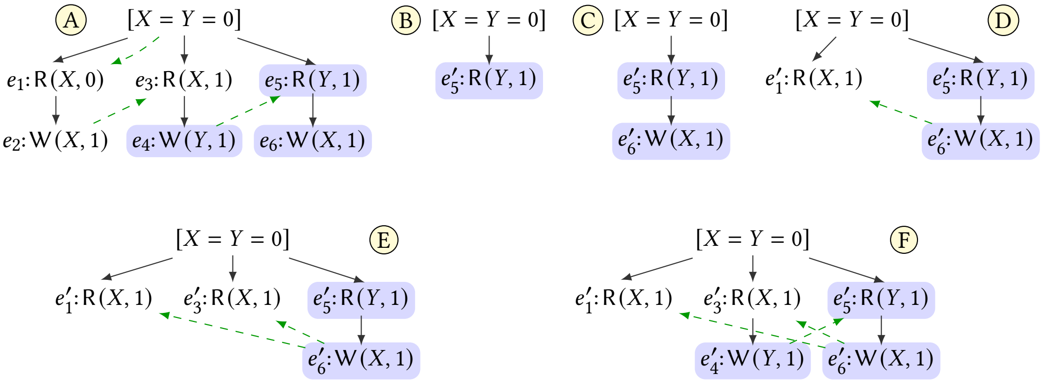


Fig. 3. XMM construction of the execution with $(\mathsf{po} \cup \mathsf{rf})$ cycle with $a = 1$ outcome for SEQ program.

unsound in the multi-execution concurrency models, including Promising semantics [Kang et al. 2017] and Weakestmo [Chakraborty and Vafeiadis 2019], as these models forbid the outcome $a = 1$ in the source SEQ program but allow it in the final transformed SEQ3 program. Although the C20 model, extended from C11 [Batty et al. 2011], supports sequentialization, it lacks DRF guarantees.

In XMM the sequentialization transformation is sound. XMM may create an execution from the SEQ program that denotes $a = 1$ as shown in Figure 3. First, we commit the events e_4, e_5, e_6 in the RC20 consistent execution in Ⓐ. Next, we re-execute the program appending the events $e'_5, e'_6, e'_1, e'_3, e'_4$ in Ⓑ to Ⓕ steps where we preserve the labels of the committed events. Once e'_4 is added, we also add the rf edge to e'_5. The resulting execution graph in Ⓕ is C20 consistent with the desired outcome $a = 1$.

3 Formal Model

In this section, we give a formal definition of the XMM, and provide the proof outlines for the soundness of program transformations and the compilation mappings, as well as the DRF theorem.

Notation. Given a binary relation A, its reflexive, transitive, reflexive-transitive closures, and inverse relation are denoted by $A^?, A^+, A^*, A^{-1}$ respectively. Relation $A_1; A_2$ denotes the composition of two relations. Relation A_{imm} is the immediate relation: $A_{\mathsf{imm}}(a, b) \triangleq A(a, b) \wedge \nexists c.\ A(a, c) \wedge A(c, b)$. Given two sets (or binary relations) X, Y, and a function $f : X \longrightarrow Y$, we use the following notations for function lifting: $f \uparrow X \triangleq \{y \in Y \mid \exists x \in X.\, f(x) = y\}$ and $f \downarrow Y \triangleq \{x \in X \mid f(x) \in Y\}$.

Definition 1. A set of *labels* LAB consists of tuples $\langle \mathsf{op}, \mathsf{loc}, \mathsf{ord}, \mathsf{val} \rangle$ of memory operation type op, accessed memory location loc, memory order ord, and read or written value val. For fence events $\mathsf{loc} = \mathsf{val} = \perp$.

Definition 2. An event $\langle \mathsf{id}, \mathsf{tid} \rangle$ is a tuple, where id is a unique identifier and tid is its thread identifier. We reserve t_{init} for the *initialization thread*, setting all memory locations to initial values.

Definition 3. An execution graph $\mathsf{G} \triangleq \langle \mathsf{E}, \mathsf{lab}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw} \rangle$ is a tuple, where E is a set of events, $\mathsf{lab} : \mathsf{E} \longrightarrow \mathsf{LAB}$ is a labeling function, $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ is the strict partial *program order*, $\mathsf{rf} \subseteq \mathsf{W} \times \mathsf{R}$ is the *reads-from* relation, $\mathsf{mo} \subseteq \mathsf{W} \times \mathsf{W}$ is the strict partial *modification order*, and $\mathsf{rmw} \subseteq \mathsf{R} \times \mathsf{W}$ is the *read-modify-write* relation. The graph is *well-formed* if the following conditions are met:

- program order is total on the subset of events of each non-initialization thread, that is, $\forall t \in \mathsf{TID} \setminus \{t_{\mathsf{init}}\}.\ \mathsf{G.po}|_t$ is total order on $\mathsf{G.E}|_t$
- program order puts initialization events before any other non-initialization events, otherwise, it orders only the events from the same thread:

$$\forall e_1, e_2.\ (\mathsf{tid}(e_1) = t_{\mathsf{init}} \wedge \mathsf{tid}(e_2) \neq t_{\mathsf{init}} \implies \langle e_1, e_2 \rangle \in \mathsf{po}) \wedge \\ (\langle e_1, e_2 \rangle \in \mathsf{po} \implies \mathsf{tid}(e_1) = t_{\mathsf{init}} \wedge \mathsf{tid}(e_2) \neq t_{\mathsf{init}} \vee \mathsf{tid}(e_1) = \mathsf{tid}(e_2))$$

- reads-from respects memory locations and values of events, that is, $\forall \langle w, r \rangle \in \mathsf{rf}.\ \mathsf{loc}(w) = \mathsf{loc}(r) \wedge \mathsf{val}(w) = \mathsf{val}(r)$
- each read event can read only from a single write event: $\forall \langle w_1, r \rangle, \langle w_2, r \rangle \in \mathsf{rf}.\ w_1 = w_2$
- reads-from the same thread has to be from a program order preceding write: $\mathsf{rf} \cap \underset{\mathsf{tid}}{=} \subseteq \mathsf{po}$.
- modification order is total on same-location write events: $\forall x \in \mathsf{loc}.\ \mathsf{G.mo}|_x$ is total
- modification order connects only the events writing to the same memory location, that is, $\forall \langle w_1, w_2 \rangle \in \mathsf{mo}.\ \mathsf{loc}(w_1) = \mathsf{loc}(w_2)$.
- read-modify-write relation connects po adjacent events with the same memory location, that is, $\mathsf{rmw} \subseteq [\mathsf{R}]; \mathsf{po}_{\mathsf{imm}}; [\mathsf{W}]$ and $\forall \langle r, w \rangle \in \mathsf{rmw}.\ \mathsf{loc}(a) = \mathsf{loc}(b)$

In the context of this paper, we assume that all execution graphs are well-formed, unless explicitly stated otherwise. We also consider only finite graphs, *i.e.,* the set of events E is always finite. We leave the case of infinite graphs, related to the problems of fairness and termination [Lahav et al. 2021], for future work.

Given a graph G and an event $e \in \mathsf{G.E}$, we write $\mathsf{G.id}(e)$, $\mathsf{G.tid}(e)$, $\mathsf{G.lab}(e)$, $\mathsf{G.op}(e)$, $\mathsf{G.loc}(e)$, $\mathsf{G.ord}(e)$, and $\mathsf{G.val}(e)$ to denote the respective components of the event. The notation $\mathsf{G.E}|_t$ denotes a set of events belonging to the thread t, and the same notation applies to a set of threads. We write G.R, G.W, and G.F to denote the set of read, write, and fence events of the graph respectively. The *behavior* of an execution graph G, denoted as $\mathsf{Behavior}(G)$, is a function mapping each location x to its final value, that is, the value written by the mo-maximal write event.

Relation G.porf unfolds into $(\mathsf{G.po} \cup \mathsf{G.rf})^+$. The *happens-before* relation hb is a memory model specific relation derived from the basic relations. It is assumed to be a partial order and a subset of program order and reads-from: $\mathsf{hb} \subseteq \mathsf{porf}$. *Viewfront* relation vf connects each write with all the events that observed it: $\mathsf{vf} \triangleq [\mathsf{W}]; \mathsf{rf}^?; \mathsf{hb}^?$.

In addition to the conventional relations defined above, we equip the execution graphs with two memory model specific relations: *relaxed program order* rpo and *stable reads-from* srf.

Relaxed program order rpo includes only those program order edges, that *must* be preserved by any reordering transformation, for instance, because of a memory fence placement. Thereby, this relation is *invariant under reordering transformations*, a fact that significantly simplifies our proofs of reordering transformations' soundness (§3.6).

Stable reads-from srf relation, while not directly used in the definition of XMM framework (§3.1), is actively employed in various proofs. It maps each read event to its *stable write event* — a write event, such that reading from it does not lead to inconsistency or a creation of a new porf cycle.

With the help of these two relations, our proofs given in §3.4, §3.5, and §3.6, can be parameterized with respect to any axiomatic memory consistency model $\mathcal{M}$ providing their *well-formed* definitions.

Definition 4. *Relaxed program order* rpo and *stable reads-from* srf relations are *well-formed*, if:

- rpo is a partial order relation, and a subset of program order: $\mathsf{rpo} \subseteq \mathsf{po}$;
- srf satisfies the same well-formedness conditions as rf, *i.e.,* it respects memory locations and values of events, and it is functional;
- srf belongs to the view-front relation, and thus cannot create new porf cycles, that is: $\mathsf{srf} \subseteq \mathsf{vf}^?; \mathsf{po} \subseteq \mathsf{porf}^?; \mathsf{po}$.

3.1 Execution Graph Construction

We next describe the operational semantics of the XMM execution graph construction. The formal rules of the semantics are presented on the Fig. 4.

Execute Rule. The first rule (Execute) corresponds to the in-order execution of an instruction. It simply adds a new event e with label ℓ to the graph and checks if the resulting graph is consistent, and *reads-from complete*. The latter property prevents the addition of thin-air reads. Thus, the (Execute) rule on itself cannot create new porf cycles.

Definition 5. Execution graph G is *reads-from complete*, denoted as *RfComplete*(G), if each read event in the graph reads from some write event belonging to the same graph: $\mathsf{G.R} \subseteq \mathsf{codom}(\mathsf{G.rf})$.

Add Event Rule. In turn, the (Add Event) rule is responsible for updating all the graph's components to accommodate for a new event. The new event e becomes the last event of the respective thread: $\Delta_{\mathsf{po}}(G, e)$. For a new read event $e \in \mathsf{W}$, a write w event to be read-from can be chosen non-deterministically: $\Delta^{\mathsf{R}}_{\mathsf{rf}}(G, w, e)$. Note that the (Add Event) rule allows omitting the

$$\frac{\mathsf{G} \xRightarrow{\langle e,\ell\rangle} \mathsf{G}' \qquad \begin{array}{c} \mathit{RfComplete}(\mathsf{G}') \\ \mathit{Consistent}(\mathcal{M}, \mathsf{G}') \end{array}}{\mathsf{G} \xrightarrow[exec]{\langle e,\ell\rangle} \mathsf{G}'} \ \text{(Execute)} \qquad \frac{\begin{array}{c} \mathsf{G}'.\mathsf{E} = \mathsf{G}.\mathsf{E} \uplus \{e\} \\ \mathsf{G}'.\mathsf{lab} = \mathsf{G}.\mathsf{lab}[e \mapsto \ell] \\ \mathsf{G}'.\mathsf{po} = \mathsf{G}.\mathsf{po} \cup \Delta_{\mathsf{po}}(\mathsf{G}, e) \\ \mathsf{G}'.\mathsf{rf} = \mathsf{G}.\mathsf{rf} \cup \Delta^{\mathsf{R}}_{\mathsf{rf}}(\mathsf{G}, w, e) \cup \Delta^{\mathsf{W}}_{\mathsf{rf}}(\mathsf{G}, R, e) \\ \mathsf{G}'.\mathsf{mo} = \mathsf{G}.\mathsf{mo} \cup \Delta_{\mathsf{mo}}(\mathsf{G}, W_1, W_2, e) \\ \mathsf{G}'.\mathsf{rmw} = \mathsf{G}.\mathsf{rmw} \cup \Delta_{\mathsf{rmw}}(\mathsf{G}, r, e) \end{array}}{\mathsf{G} \xRightarrow{\langle e,\ell\rangle} \mathsf{G}'} \ \text{(Add Event)}$$

$$\frac{\begin{array}{c} C \subseteq \mathsf{G}'.\mathsf{E} \qquad \mathcal{D} \subseteq \mathsf{f} \uparrow C \qquad \mathsf{G}.\mathsf{E} = \mathcal{D} \cup \mathsf{G}.\mathsf{E}|_{\mathcal{T}} \\ \mathcal{D} \text{ is } \mathsf{G}.\mathsf{po} \text{ maximal } \textit{s.t.}\ \mathsf{dom}(\mathsf{G}.\mathsf{po}; [\mathcal{D}]) \subseteq \mathcal{D} \\ \mathsf{dom}(\mathsf{G}'.\mathsf{rpo}; [\mathsf{G}'.\mathsf{E} \setminus \mathcal{D}]) \subseteq \mathcal{D} \\ \mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}}, \mathsf{G}', C) \\ \langle \mathsf{G}', C \rangle \vdash \mathsf{G}|_{\mathcal{D}} \Rightarrow^{*} \mathsf{G}' \end{array} \qquad \begin{array}{c} \mathit{Consistent}(\mathcal{M}, \mathsf{G}') \\ \mathit{CommitEmbedded}(\mathsf{G}, \mathsf{G}', C, \mathsf{f}) \\ \mathit{ThreadOrderedUEvents}(\mathsf{G}', C, \preceq_{\mathsf{tid}}) \end{array}}{\mathsf{G} \xrightarrow[re-exec]{C} \mathsf{G}'} \ \text{(Re-Execute)}$$

$$\frac{\mathsf{G}_i \xRightarrow{\langle e,\ell\rangle} \mathsf{G}_{i+1} \qquad \mathit{WellFormed}(\mathsf{G}_{i+1}, \mathsf{G}', C)}{\langle \mathsf{G}', C \rangle \vdash \mathsf{G}_i \xRightarrow{\langle e,\ell\rangle} \mathsf{G}_{i+1}} \ \text{(Guided Step)}$$

$$\begin{array}{llll}
\Delta_{\mathsf{po}}(\mathsf{G}, e) & \triangleq (\mathsf{G}.\mathsf{E}_{\mathsf{init}} \cup \mathsf{G}.\mathsf{E}_{e.\mathsf{tid}}) \times \{e\} & & \\
\Delta^{\mathsf{R}}_{\mathsf{rf}}(\mathsf{G}, w, r) & \triangleq \{\langle w, r\rangle\} & \textbf{if } r \in \mathsf{R} \land \bot \neq w \in \mathsf{G}.\mathsf{W} & \textbf{otherwise } \emptyset \\
\Delta^{\mathsf{W}}_{\mathsf{rf}}(\mathsf{G}, R, w) & \triangleq \{w\} \times R & \textbf{if } w \in \mathsf{W} \land R \subseteq \mathsf{G}.\mathsf{R} & \textbf{otherwise } \emptyset \\
\Delta_{\mathsf{mo}}(\mathsf{G}, W_1, W_2, w) & \triangleq W_1 \times \{w\} \cup \{w\} \times W_2 & \textbf{if } w \in \mathsf{W} \land W_1, W_2 \subseteq \mathsf{G}.\mathsf{W}_{\mathsf{loc}(w)} & \textbf{otherwise } \emptyset \\
\Delta_{\mathsf{rmw}}(\mathsf{G}, r, w) & \triangleq \{\langle r, w\rangle\} & \textbf{if } w \in \mathsf{W} \land \bot \neq r \in \mathsf{G}.\mathsf{R} \land \mathsf{IsExclusiveWrite}(w) & \textbf{otherwise } \emptyset
\end{array}$$

Fig. 4. Rules of the XMM execution graph construction

addition of a reads-from edge for the new read event by choosing $w = \bot$. Although such a graph would be immediately discarded by the $\mathit{RfComplete}(\mathsf{G}')$ constraint of the (Execute) rule, this possibility will become handy later when we define the re-execution semantics. For a new write event $e \in \mathsf{W}$, a set of reads R is chosen non-deterministically to create outgoing reads-from edges: $\Delta^{\mathsf{W}}_{\mathsf{rf}}(G, R, e)$. In this case, the modification order is also updated by inserting the new event in-between two subsets W_1, W_2 of the write events to the same location: $\Delta_{\mathsf{mo}}(G, W_1, W_2, e)$. Finally, if e is additionally an exclusive write originated from a read-modify-write operation, it is connected with the preceding read event to form a new read-modify-write edge: $\Delta_{\mathsf{rmw}}(G, r, e)$.

Re-Execute Rule. Next, the (Re-Execute) rule formalizes the re-execution semantics. It is parametrized by a set of committed events $C \subseteq \mathsf{G}'.\mathsf{E}$ that should be preserved during re-execution. A set of *determined* events $\mathcal{D}$ is a program order maximal prefix-closed subset of committed events. To apply the rule, one has to choose a partial event mapping $\mathsf{f} : \mathsf{G}'.\mathsf{E} \rightharpoonup \mathsf{G}.\mathsf{E}$, and some strict partial order on thread identifiers $\prec_{\mathsf{tid}}$. Then the program is re-executed starting from the graph G restricted to the subset of determined events.

Let $\mathcal{T}$ be the set of re-executed threads: $\mathcal{T} \triangleq \{t \mid \exists e \in (\mathsf{G}'.\mathsf{E} \setminus \mathcal{D}).\ \mathsf{G}'.\mathsf{tid}(e) = t\}$. We require the original graph to contain only determined events or events from the re-executed threads. Also note that it is prohibited for the events in $\mathsf{G}'.\mathsf{E} \setminus \mathcal{D}$ to induce new rpo edges. Finally, the resulting graph G' is checked to be consistent. The meaning of another two predicates *CommitEmbedded* and *ThreadOrderedUEvents* is explained later in this section.

Guided Step Rule. Consider that during a re-execution, the addition of new events *is guided by* the desired events in the final graph G'. The (Guided Step) rule captures this. It is additionally parameterized by the final graph G' together with the subset of committed events $\mathcal{C}$. Whenever a new event is added to the graph G_i using the (Add Event) rule, the resulting tuple $\langle \mathsf{G}_{i+1}, \mathsf{G}', \mathcal{C} \rangle$ is checked to be *well-formed* in the following sense.

Definition 6. A configuration $\langle \mathsf{G}, \mathsf{G}', \mathcal{C} \rangle$ is *well-formed* if the following conditions are met:

- graph G is well-formed: $\mathit{WellFormed}(\mathsf{G})$
- two graphs coincide on the subset of committed events: $\mathsf{G}'|_{\mathsf{G.E} \cap \mathcal{C}} = \mathsf{G}|_{\mathsf{G.E} \cap \mathcal{C}}$
- the graph G' is reads-from complete on the subset of committed events: $\mathit{RfComplete}(\mathsf{G}'|_{\mathcal{C}})$
- all reads from the graph G read from some write event in G, or otherwise committed: $\mathsf{G.R} \subseteq \mathrm{codom}(\mathsf{G.rf}) \cup \mathcal{C}$.

Most importantly, the last condition prevents the addition of thin-air reads by asserting that a newly added read event can be set to temporarily read-from "nowhere", *but only as long as it is committed.* In combination with the requirement on the final graph to be reads-from complete on the subset of committed events, this property guarantees that all the reads will eventually be restored to read-from some well-defined write event belonging to the graph.

The well-formedness constraint provides sufficient conditions for the re-execution to be possible. Intuitively, the relation $\mathsf{G}'.\mathsf{po} \cup \mathsf{G}'.\mathsf{rf}; [\mathcal{U}]$ has to be acyclic, where $\mathcal{U} \triangleq \mathsf{G}'.\mathsf{E} \setminus \mathcal{C}$ is a set of uncommitted events. Thus, whenever an uncommitted read is added to the graph, its respective reads-from write is already in the graph. The following lemma formalizes this intuition.

Lemma 1. *Let* G *and* G' *be two execution graphs, such that* G *is a program order prefix-closed subgraph of* G'*:* $\mathsf{G}'|_{\mathsf{G.E}} = \mathsf{G}$ *and* $\mathrm{dom}(\mathsf{G.po}; [\mathsf{G.E}]) \subseteq \mathsf{G.E}$*. Let* $\mathcal{C} \subseteq \mathsf{G}'.\mathsf{E}$ *be a subset of committed events,* $\mathcal{U} \triangleq \mathsf{G}'.\mathsf{E} \setminus \mathcal{C}$ *be a subset of uncommitted events, and* $\Delta = \mathsf{G}'.\mathsf{E} \setminus \mathsf{G.E}$ *be a difference of the two graphs' event sets. Moreover, assume* $\mathit{WellFormed}(\mathsf{G}, \mathsf{G}', \mathcal{C})$ *holds.*

Let $\ell \triangleq [e_1, \ldots, e_n]$ *be an enumeration of a set* Δ*, such that the following conditions are met:*

- $\mathsf{G}'.\mathsf{po} \cup \mathsf{G}'.\mathsf{rf}; [\mathcal{U}]$ *respects the order induced by* ℓ*:* $\langle e_i, e_j \rangle \in \mathsf{G}'.\mathsf{po} \cup \mathsf{G}'.\mathsf{rf}; [\mathcal{U}]$ *implies* $i < j$
- Δ *is* rf*-complete:* $\forall r \in \mathsf{R} \cap \Delta.\ \exists w \in \mathsf{G.W} \cup \Delta.\ \langle w, r \rangle \in \mathsf{G}'.\mathsf{rf}$

Then, the graph G' *can be constructed from* G *as follows:* $\langle \mathsf{G}', \mathcal{C} \rangle \vdash \mathsf{G} \overset{\ell}{\Rightarrow}^{*} \mathsf{G}'$.

We provide the proof of this lemma, as well as other lemmas stated in this section, in Appendix A.

Subgraph embedding. The $\mathit{CommitEmbedded}(\mathsf{G}, \mathsf{G}', \mathcal{C}, \mathsf{f})$ predicate ensures that the original graph G and the resulting graph G' share a common subgraph consisting of committed events. In essence, we want to say that $\mathsf{G}|_{\mathcal{C}} = \mathsf{G}'|_{\mathcal{C}}$. Yet technically, the two graphs may be formed by two disjoint sets of events. Therefore, instead of simple event-wise equality, we use a *partial event mapping function* $\mathsf{f} : \mathsf{G}'.\mathsf{E} \rightharpoonup \mathsf{G.E}$ which re-enumerates events accordingly.

Definition 7. Given two graphs G, G', a set of committed events $\mathcal{C} \subseteq \mathsf{G}'.\mathsf{E}$, and an injective function $\mathsf{f} : \mathsf{G}'.\mathsf{E} \rightharpoonup \mathsf{G.E}$, we say that committed subgraph of G' *is embedded into* graph G under f, if f preserves labels, relaxed program order, reads-from, modification order, and read-modify-write relation of committed events:

$$\forall c \in \mathcal{C}.\ \mathsf{f}(c) \neq \bot \land\ \mathsf{G}'.\mathsf{lab}(c) = \mathsf{G.lab}(\mathsf{f}(c)) \quad | \quad \mathsf{f} \uparrow \mathsf{G}'.X|_{\mathcal{C}} = \mathsf{G}.X|_{\mathsf{f} \uparrow \mathcal{C}} \text{ for } X \in \{\mathsf{rpo}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw}\}$$

Restricting uncommitted events. Finally, the predicate $\mathit{ThreadOrderedUEvents}(\mathsf{G}', C, \prec_{\mathsf{tid}})$ imposes an additional restriction on the uncommitted events. As demonstrated later in §3.6, it is essential for ensuring the soundness of transformations.

Definition 8. Given a graph G' and a set of committed events $C \subseteq \mathsf{G}'.\mathsf{E}$, we say that *uncommitted events respect the thread ordering relation* $\prec_{\mathsf{tid}}$, if the view-front relation incoming into a thread with uncommitted events respects the thread ordering:

$$\mathsf{G}'.\mathsf{vf}; \mathsf{G}'.\mathsf{tid} \downarrow \underset{\mathsf{tid}}{=}; [\mathcal{U}] \subseteq \mathsf{G}'.\mathsf{tid} \downarrow \preceq_{\mathsf{tid}}.$$

In particular, this implies that each uncommitted read either reads from a program order preceding write, or from a write event in a $\prec_{\mathsf{tid}}$ preceding thread: $\mathsf{rf}; [\mathcal{U}] \subseteq \mathsf{G}'.\mathsf{po} \cup \mathsf{G}'.\mathsf{tid} \downarrow \prec_{\mathsf{tid}}$.

Using this constraint, we show that the re-execution can always proceed thread-by-thread in accordance with the chosen thread ordering $\prec_{\mathsf{tid}}$. The next lemma assures that the events can be reordered in the re-execution sequence, so that they become grouped by threads, while also respecting the $\prec_{\mathsf{tid}}$ relation.

Lemma 2. *Let $\mathsf{G}, \mathsf{G}', C, \mathcal{U}, \Delta, \ell$ be defined similarly as in Lemma 1. In particular, we have that $\mathit{WellFormed}(\mathsf{G}, \mathsf{G}', C)$. Moreover, suppose the predicate $\mathit{ThreadOrderedUEvents}(\mathsf{G}', C, \prec_{\mathsf{tid}})$ holds, restricting the uncommitted events for a given $\prec_{\mathsf{tid}}$ relation.*

Suppose ℓ' is the list containing the same events, but reordered and grouped by threads according to $\prec_{\mathsf{tid}}$ relation (below $\ell_{t_{i_k}}$ is a sublist of ℓ containing the events from thread t_{i_k}):

$$\ell' \triangleq \ell_{t_{i_1}} \cdot \ldots \cdot \ell_{t_{i_n}} \quad \mid \quad k < m \implies t_{i_k} \prec_{\mathsf{tid}} t_{i_m} \qquad \textit{or, in other words:}$$

$$\ell' \quad \textit{is ordered by} \quad \mathsf{G}'.\mathsf{po} \cup \mathsf{G}'.\mathsf{tid} \downarrow \prec_{\mathsf{tid}}$$

Then, the resulting list ℓ' can still be used to re-execute the graph leading to the same result:

$$\langle \mathsf{G}', C \rangle \vdash \mathsf{G} \overset{\ell}{\Rightarrow}^{*} \mathsf{G}' \quad \textit{if and only if} \quad \langle \mathsf{G}', C \rangle \vdash \mathsf{G} \overset{\ell'}{\Longrightarrow}^{*} \mathsf{G}'.$$

Putting everything together. The XMM graph construction on each step either appends one single event to the existing graph, or it non-deterministically selects a subset of committed events and re-builds the graph from scratch, preserving the committed subgraph intact. In what follows, we will use the notation $\mathsf{G} \longrightarrow \mathsf{G}'$ to indicate an XMM construction step, which is either an execution or a re-execution step. Any graph that can be constructed from an initial graph using the XMM construction procedure is called XMM consistent with respect to the memory consistency model $\mathcal{M}$. The resulting memory model is denoted as $\mathsf{XMM}(\mathcal{M})$.

3.2 Reconciling Execution Graphs with Operational Semantics

Next, we describe how to reconnect the execution graphs with the operation semantics of individual threads given in terms of a labeled transition system. We decouple the process of graph construction from the threads' operational semantics by putting a constraint that ties-in an execution graph and *traces* generated by the abstract machine.

We impose only one requirement to the labeled transition system — in addition to the operation's label, the transition relation should be annotated with the *instruction identifier.*

Let InstrID be a set of abstract *instruction identifiers.* Then for a graph G we can consider a function $\mathcal{I} : \mathsf{G}.\mathsf{E} \longrightarrow \mathsf{InstrID}$ which maps an event to an instruction which produced that event. In the context of this paper, we assume that $\mathcal{I}$ is always an injective function, that is every instruction corresponds to at most one event. In other words, we consider only loop-free programs (or programs where all loops were unrolled to a certain bound). Because we only consider finite graphs and leave the questions of program's termination out of scope, these assumptions do not entail any new limitations of our approach.

Definition 9. We assume that for a program P the thread-local operational semantics is given as a *labeled transition system* L of the form $\triangleq \langle \mathsf{S}, \sigma_{\mathsf{init}}, \longrightarrow \rangle$, where

- S — is a set of states,
- $\sigma_{\mathsf{init}} : \mathsf{TID} \longrightarrow \mathsf{S}$ — is an initial states thread mapping,
- $\longrightarrow\ \subseteq \mathsf{S} \times (\mathsf{InstrID} \times \mathsf{LAB}_{\epsilon}) \times \mathsf{S}$ — is a transition relation, and
- $\mathsf{LAB}_{\epsilon} \triangleq \mathsf{LAB} \uplus \{\epsilon\}$ — is a set of labels augmented with the empty label.

We write $\sigma \xrightarrow{i:\ell} \sigma'$ to denote a transition of the operational semantics.

Definition 10. Given a labeled transition system and a graph G let $\mathsf{G.T}(t) = [e_1, \ldots, e_n]$ be a program-order sorted sequence of thread's t events, that is a sequence, *s.t.* $\mathsf{G.T}(t) = \mathsf{G.E}|_t$ and $e_1 \xrightarrow{\mathsf{po}} e_2 \xrightarrow{\mathsf{po}} \ldots \xrightarrow{\mathsf{po}} e_n$. We say that the thread t of the graph G *conforms to a trace of the operational semantics*, denoted as *TraceConforming*$(\mathsf{L}, \mathsf{G}, t)$, if the sequence of labels corresponding to the thread's events forms a valid trace, that is:

$$\sigma_{\mathsf{init}}(t) \xrightarrow{\iota_1:\epsilon}{}^{*} \xrightarrow{\iota_1:\mathsf{lab}(e_1)} \xrightarrow{\iota_1:\epsilon}{}^{*} \sigma_1 \xrightarrow{\iota_2:\epsilon}{}^{*} \xrightarrow{\iota_2:\mathsf{lab}(e_2)} \xrightarrow{\iota_2:\epsilon}{}^{*} \ldots \xrightarrow{\iota_n:\epsilon}{}^{*} \xrightarrow{\iota_n:\mathsf{lab}(e_n)} \xrightarrow{\iota_n:\epsilon}{}^{*} \sigma_n.$$

If *TraceConforming*$(\mathsf{L}, \mathsf{G}, t)$ is true for every thread t of the graph G, we say that the whole graph *conforms to the operational semantics*, written as *TraceConforming*(L, G).

Finally, the step of the combined semantics simply performs a graph construction step assuring that the resulting graph conforms to the operational semantics:

$$\frac{\mathsf{G} \longrightarrow \mathsf{G}' \qquad \mathit{TraceConforming}(\mathsf{L}, \mathsf{G}')}{\mathsf{P}, \mathsf{L} \vdash \mathsf{G} \longrightarrow \mathsf{G}'}$$

3.3 XC20 Consistency

We instantiate the $\mathsf{XMM}(\mathcal{M})$ framework with the C/C++20 memory consistency to derive a model we call XC20. We start with the definition of RC20 from Lahav et al. [2017], minus porf acyclicity. We then extend it with two novel notions of *relaxed program order* and *relaxed happens-before*. As was already mentioned, the main motivation behind the introduction of these relations is to simplify the proofs of reordering transformations' soundness — unlike their conventional counterparts, these two relations are *invariant under reordering transformations*. We therefore reformulate C/C++20 consistency constraints in terms of relaxed program order and relaxed happens-before, and show that the two definitions are equivalent.

We do not model sequentially consistent accesses and fences in XC20, following the definition of RC20 and the observations from [Margalit and Lahav 2021]: very few practical algorithms employ SC accesses and become incorrect when release/acquire accesses are used instead; as for the SC fences, in RC20 they can be represented by acquire-release fences and a read-modify-write operation to a distinguished memory location. This is not a fundamental limitation as the `srf` relation can also be refined to account for SC accesses [Chakraborty and Vafeiadis 2019] in future extensions of the model.

Figure 5 formally defines the following relations.

- *Synchronizes-with relation* sw connects synchronized events, such as a release write and an acquire read that reads-from it.
- *Happens-before order* hb is a transitive closure of program order and synchronizes-with relations.
- *From-reads* relation fr, connects a read event to all the write events, that "overwrite" the write it reads-from. Relation *extended coherence order* eco is transitive closure of rf, mo, and fr relations.

$$
\begin{aligned}
\mathsf{sw} &\triangleq [\mathsf{E}^{\textsc{rel}\sqsubseteq}]; ([\mathsf{F}]; \mathsf{po})^?; [\mathsf{W}^{\textsc{rlx}\sqsubseteq}]; (\mathsf{rf}; \mathsf{rmw})^*; \mathsf{rf}; [\mathsf{R}^{\textsc{rlx}\sqsubseteq}]; (\mathsf{po}; [\mathsf{F}])^?; [\mathsf{E}^{\textsc{acq}\sqsubseteq}] \\
\mathsf{hb} &\triangleq (\mathsf{po} \cup \mathsf{sw})^+ \qquad \mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{mo} \qquad \mathsf{eco} \triangleq (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr})^+ \\
\mathsf{rpo} &\triangleq ([\mathsf{R}^{\textsc{rlx}\sqsubseteq}]; \mathsf{po}; [\mathsf{F}^{\textsc{acq}\sqsubseteq}] \cup [\mathsf{E}^{\textsc{acq}\sqsubseteq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{E}^{\textsc{rel}\sqsubseteq}] \cup [\mathsf{F}^{\textsc{rel}\sqsubseteq}]; \mathsf{po}; [\mathsf{W}^{\textsc{rlx}\sqsubseteq}])^+ \\
\mathsf{rhb} &\triangleq (\mathsf{po}|_{\mathsf{loc}} \cup \mathsf{rpo} \cup \mathsf{sw})^+
\end{aligned}
$$

$$
\mathrm{C20} \triangleq \begin{cases} \mathsf{hb}; \mathsf{eco}^? \text{ is irreflexive.} \\ \mathsf{rmw} \cap (\mathsf{fr}; \mathsf{mo}) \text{ is empty.} \end{cases}
\qquad
\mathrm{C20}_{\mathsf{rhb}} \triangleq \begin{cases} \mathsf{rhb}; \mathsf{eco}^? \text{ is irreflexive.} \\ \mathsf{rmw} \cap (\mathsf{fr}; \mathsf{mo}) \text{ is empty.} \end{cases}
\qquad
\begin{matrix} \text{(Coherence)} \\ \text{(Atomicity)} \end{matrix}
$$

$$
\mathrm{RC20} \triangleq \mathrm{C20} \text{ and } (\mathsf{po} \cup \mathsf{rf}) \text{ is acyclic} \quad \text{(No-OOTA)}
\qquad
\mathrm{XC20} \triangleq \mathrm{XMM}(\mathrm{C20}_{\mathsf{rhb}})
$$

Fig. 5. RC20 and XC20 consistency axioms. C20 model refers to RC20 minus the (No-OOTA) axiom.

- *Relaxed program order* rpo is a partial order relation, a subset of program order, derived from the memory fences placement in a program.
- *Relaxed happens-before* rhb is a subset of happens-before relation, that is defined similarly as happens-before order, but uses union of same-location and relaxed program orders in-place of the program order.

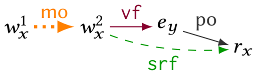
Proposition 1. *The following statements are true: (1)* $\mathsf{po}; \mathsf{sw} = \mathsf{rpo}; \mathsf{sw}$ *and* $\mathsf{sw}; \mathsf{po} = \mathsf{sw}; \mathsf{rpo}$, *(2)* $\mathsf{hb} = \mathsf{po} \cup \mathsf{rhb}$, *(3)* $\mathsf{hb}|_{\mathsf{loc}} = \mathsf{rhb}|_{\mathsf{loc}}$, *(4)* $\mathsf{hb}|_{\neq \mathsf{tid}} = \mathsf{rhb}|_{\neq \mathsf{tid}}$.

Axioms of the C20 consistency model are also given on Figure 5. RC20 model is simply C20 plus $(\mathsf{po} \cup \mathsf{rf})$ acyclicity constraint. We define XC20 model as $\mathrm{XMM}(\mathrm{C20}_{\mathsf{rhb}})$, where $\mathrm{C20}_{\mathsf{rhb}}$ is a C20 variant that uses rpo and rhb definitions in-place of po and hb respectively.

Lemma 3. C20 *and* $\mathrm{C20}_{\mathsf{rhb}}$ *consistency models are equivalent.*

As can be seen, the (Atomicity) constraint does not involve the happens-before relation, and thus it does not change in $\mathrm{C20}_{\mathsf{rhb}}$ Equivalence for the (Coherence) axiom follows from the fact that extended coherence order connects only the same-location events. Therefore, in the original coherence definition we can substitute hb with $\mathsf{hb}|_{\mathsf{loc}}$, which is equal to $\mathsf{rhb}|_{\mathsf{loc}}$ due to Proposition 1.

Finally, we define the *stable reads-from* relation for C20, using an auxiliary *viewfront* relation.



Definition 11. The *stable reads-from* relation maps each read to the latest in modification order observed write: $\mathsf{srf} \triangleq (\mathsf{vf}; \mathsf{po}; [\mathsf{R}])|_{\neg \mathsf{loc}} \setminus (\mathsf{mo}; \mathsf{vf}; \mathsf{po})$.

Consider the example execution where $\langle w_x^1, w_x^2 \rangle \in \mathsf{mo}$, $\langle w_x^2, e_y \rangle \in \mathsf{vf}$ where $x \neq y$, and $\langle e_y, r \rangle \in \mathsf{po}$ hold. In this case, $\langle w_x^2, r_x \rangle \in \mathsf{srf}$ holds and r_x can read from w_x^2. Note that w_x^1 is not in srf relation with r_x as $\langle w_x^1, r_x \rangle \in \mathsf{mo}; \mathsf{vf}; \mathsf{po}$ holds.

3.4 Data Race Freedom Guarantees

We next show that XMM adheres to the *data race freedom (*DRF*) guarantees.* Recall that a DRF guarantee allows considering only the behavior of a program under a stronger memory model, for example, *sequential consistency model* (SC), provided that the said program is race-free under this stronger model.

Definition 12. Two concurrent events are in *race* if they access the same location, and at least one of them is a write event. Let G.Race be the set of all racy events of execution G and let G.Race$(o) \subseteq$ G.Race be the set of races where at least one of the involved events has memory order weaker or equal to o.

$$\mathsf{Race} \triangleq (\mathsf{E} \times \mathsf{E} \cap =_{\mathsf{loc}} \cap \mathsf{one}(\mathcal{W})) \setminus (\mathsf{hb}^{?} \cup \mathsf{hb}^{-1}) \text{ and } \mathsf{Race}(o) \triangleq \mathsf{Race} \cap \mathsf{one}(\mathsf{E}_{\sqsubseteq o})$$
$$\text{where } \langle x, y\rangle \in \mathsf{one}(A) \triangleq (x \in A \vee y \in A)$$

Definition 13. We say that a memory model $\mathcal{M}$ provides the *data-race freedom guarantee* with respect to a stronger memory model $\mathcal{M}'$ and memory order o, if for every program P, such that all $\mathcal{M}'$-consistent execution graphs of P are o-race-free, the memory model $\mathcal{M}$ also assigns only $\mathcal{M}'$-consistent graphs to P:

$$\forall \mathsf{P}.\ (\forall \mathsf{G} \in [\![\mathsf{P}]\!]_{\mathcal{M}'}.\ \mathsf{G.Race}(o) = \emptyset) \implies [\![\mathsf{P}]\!]_{\mathcal{M}} = [\![\mathsf{P}]\!]_{\mathcal{M}'}.$$

We denote this fact as $\mathcal{M} \in \mathsf{DRF}^{o}(\mathcal{M}')$, and we omit the memory access mode when its name matches the name of the memory model itself (as in case of the *sequential consistency* DRF(SC) and *release-acquire consistency* DRF(RA)).

The following lemma is the key component of the proof. It states that for a race-free program, the XC20 operational semantics can only produce porf acyclic execution graphs.

Lemma 4. *Let* P *be a program, such that all of its* RC20*-consistent execution graphs are* relax (RLX) race-free. *Suppose additionally that* G *is a* RC20 *consistent graph of this program. Then every graph* G′*, such that* $\mathsf{G} \longrightarrow \mathsf{G}'$*, is also* porf *acyclic.*

We note that the (Execute) rule on itself cannot create a new porf cycle, and thus the only interesting case is the (Re-Execute) rule: $\mathsf{G} \xrightarrow[re-exec]{} \mathsf{G}'$. For this case, we consider a *re-execution frontier* $\mathcal{F}$ that is a set of events from G′ immediately succeeding determined events $\mathcal{D}$ in program order. We show that without loss of generality, one can assume all the events in the set $\mathcal{F}$ are uncommitted relaxed reads, which read-from a different write event compared to the one they read-from in G. Suppose r is the first re-executed read r from the frontier, a write w_1 from which it reads in the original graph G, and a write w_2 from which it reads in the re-executed graph G′. Considering these three events, we demonstrate that we can always arrive at a contradiction with the consistency of either graph G or G′, or we can construct a data-race in G, arriving at contradiction as P is relax race-free.

A more detailed proof can be found in Appendix C.

Theorem 1. XC20 *model provides* $\mathsf{DRF}^{\mathit{RLX}}(\mathsf{RC20})$*, as well as* DRF(RA) *and* DRF(SC) *guarantees.*

Proof. The fact that XC20 provides $\mathsf{DRF}^{\mathsf{RLX}}(\mathsf{RC20})$ guarantee is a simple corollary of the Lemma 4. Indeed, XC20 is simply XMM(C20), C20 is write-read coherent, and RC20-consistency is just C20-consistency augmented with the porf acyclicity.

Next, let's prove that XC20 ∈ DRF(RA). Indeed, if a program P is ACQ-REL-race-free, it is also RLX-race-free, and thus $[\![\mathsf{P}]\!]_{\mathsf{XC20}} = [\![\mathsf{P}]\!]_{\mathsf{RC20}}$ by $\mathsf{XC20} \in \mathsf{DRF}^{\mathsf{RLX}}(\mathsf{RC20})$. Then, due to RC20 ∈ DRF(RA) we have $[\![\mathsf{P}]\!]_{\mathsf{RC20}} = [\![\mathsf{P}]\!]_{\mathsf{RA}}$, and thus finally $[\![\mathsf{P}]\!]_{\mathsf{XC20}} = [\![\mathsf{P}]\!]_{\mathsf{RA}}$.

Similar reasoning applies to prove that XC20 ∈ DRF(SC). □

3.5 Soundness of Compilation Mappings

Further, we show that XC20 supports optimal compilation mappings to the hardware memory models of x86-TSO, ARM, and Power. The optimal compilation mapping is the one that does not introduce unnecessary memory fences — in particular, compiling relaxed accesses without fences.

To this end, we leverage the *intermediate memory model* IMM [Podkopaev et al. 2019], which encapsulates the implementation details of particular hardware memory models, providing a convenient abstraction for compilation correctness proofs. IMM already supports the same memory order modes as C/C++, providing their optimal compilation mappings to hardware-specific instructions and fences, and the soundness of these compilation mappings is well-established. Therefore, for the XC20-to-IMM compilation, the mapping is essentially an identity mapping, leaving us to prove that every IMM-consistent execution graph is also XC20-consistent.

Theorem 2. *Let* P *be a program, and* G *be its* IMM *consistent execution graph. Then* G *is also an* XC20 *consistent execution graph.*

To prove this theorem, we show that using the XMM operational semantics, it is possible to construct a series of execution graphs $G_0 \rightarrow^* G_1 \ldots \rightarrow^* G_n \rightarrow^* G_{n+1}$, where $G_0 = G_{\mathsf{init}}$ and $G_{n+1} = G$, such that all the graphs in this sequence are C20 consistent. Note that IMM consistency implies C20 consistency, so it is sufficient to show that all graphs are IMM consistent.

We use the fact that under the IMM model, there exists a global *causality order* on all the events in a graph, given by the ar relation (see Appendix D for a formal definition). The ar relation includes, among others, ordering constraints arising due to the placement of memory fences and *syntactic dependencies* between instructions, such as *data*, *control*,, or *address* dependencies.

For the given IMM consistent graph G, we order all its read events G.R according to the ar relation, deriving a list of read events $[r_1, \ldots, r_n]$. We then define each graph G_i in the sequence to contain all the events of G, except those that have control or address dependency on r_i, and, moreover, we redirect r_i to read from a stable write using srf relation. We then show that for each pair of graphs G_i and G_{i+1}, a re-execution step can be made: $G_i \xrightarrow[re-exec]{C} G_{i+1}$, with the set of commit events C being defined as all events in G_i, except those that follow r_i in causal order ar: $C \triangleq G_i.\mathsf{E} \setminus \mathsf{codom}([r_i]; \mathsf{ar})$. By definition of ar, it follows that changing the reads-from source for a read event r_i can only affect events in its causal suffix, and thus, indeed, all the committed events can be re-executed safely. Finally, it remains to notice that in the graph G_1 all reads are reading from a stable write. Because $\mathtt{srf} \subseteq \mathsf{porf}^?; \mathsf{po}$ we have that G_1 is porf acyclic graph, implying that it can be constructed from the initial graph using only the in-order execution steps: $G_{\mathsf{init}} \xrightarrow[exec]{}^* G_1$.

A more detailed proof can be found in Appendix D.

3.6 Soundness of Program Transformations

In this section, we outline the proofs of the program transformation's soundness. A program transformation *tr* is a code rewriting rule, mapping a source program P_{src} into a target program P_{tgt}: $P_{\mathsf{src}} \overset{tr}{\rightsquigarrow} P_{\mathsf{tgt}}$. A program transformation is *sound* if it does not introduce new behaviors.

We consider three types of transformations: *redundant access elimination*, *reordering of independent instructions*, and *thread sequentialization*.

In the context of the XMM memory model, the soundness of program transformations boils down to the following definition.

Definition 14. A program transformation $P_{\mathsf{src}} \overset{tr}{\rightsquigarrow} P_{\mathsf{tgt}}$ is *sound* in the $\mathsf{XMM}(\mathcal{M})$ memory model, if for every $\mathsf{XMM}(\mathcal{M})$-consistent execution graph G_t of P_{tgt} there exists $\mathsf{XMM}(\mathcal{M})$-consistent execution graph G_s of P_{src} with the same behavior.

For all the aforementioned program transformations, our proofs follow the same structure based on the standard *simulation technique*. For each transformation we define a simulation relation $\mathcal{R}(G_s, G_t, \mathsf{m})$ between the source graph, the target graph, and the event mapping function $\mathsf{m} : G_t.\mathsf{E} \longrightarrow G_s.\mathsf{E}$. We next prove the following lemmas:

$a \downarrow b \longrightarrow$	$\mathsf{R}_{\sqsubseteq\text{ACQ}}$	W_{NA}	W_{RLX}	$\mathsf{W}_{\sqsupseteq\text{REL}}$	F_{ACQ}	F_{REL}	$\mathsf{F}_{\text{ACQ-REL}}$
R_{NA}	✓	✓	✓	×	✓	×	×
R_{RLX}	✓	✓	✓	×	×	×	×
$\mathsf{R}_{\sqsupseteq\text{ACQ}}$	×	×	×	×	✓	×	×
$\mathsf{W}_{\sqsubseteq\text{REL}}$	✓	✓	✓	×	✓	×	×
F_{ACQ}	×	×	×	×	=	×	×
F_{REL}	✓	✓	×	✓	✓	=	×
$\mathsf{F}_{\text{ACQ-REL}}$	×	×	×	×	×	×	=

(a) Reordering of accesses $a; b \rightsquigarrow b; a$ where a and b accesses different memory locations.

$$\mathsf{R}_o(X, v); \mathsf{R}_{o'}(X, v') \rightsquigarrow \mathsf{R}_o(X, v) \quad \text{(RAR)}$$

$$\mathsf{W}_o(X, v); \mathsf{R}_{o'}(X, v') \rightsquigarrow \mathsf{W}_o(X, v) \quad \text{(RAW)}$$

$$\mathsf{W}_{o'}(X, v'); \mathsf{W}_o(X, v) \rightsquigarrow \mathsf{W}_o(X, v) \quad \text{(OW)}$$

$$\{\mathsf{F}_o; \mathsf{F}_{o'}\} \text{ or } \{\mathsf{F}_{o'}; \mathsf{F}_o\} \rightsquigarrow \mathsf{F}_o \quad \text{(FE)}$$

(b) Eliminations where $o' \sqsubseteq o$.

$$T_1 \mid\mid T_2 \rightsquigarrow T_1; T_2$$

(c) Sequentialization

Fig. 6. Safe transformations.

- initial source graph simulates initial target graph: $\mathcal{R}(\mathsf{G}_\mathsf{s}^{\mathsf{init}}, \mathsf{G}_\mathsf{t}^{\mathsf{init}}, \mathsf{m}^{\mathsf{init}})$;
- for any source and target graphs, *s.t.* $\mathcal{R}(\mathsf{G}_\mathsf{s}, \mathsf{G}_\mathsf{t}, \mathsf{m})$ holds, whenever target graph performs an $\mathsf{XMM}(\mathcal{M})$ step $\mathsf{G}_\mathsf{t} \longrightarrow \mathsf{G}_\mathsf{t}'$, source graph can also take zero or multiple $\mathsf{XMM}(\mathcal{M})$ steps $\mathsf{G}_\mathsf{s} \longrightarrow^* \mathsf{G}_\mathsf{s}'$, *s.t.* $\mathcal{R}(\mathsf{G}_\mathsf{s}', \mathsf{G}_\mathsf{t}', \mathsf{m}')$ holds again for some event mapping m';
- for any source and target graphs, *s.t.* $\mathcal{R}(\mathsf{G}_\mathsf{s}, \mathsf{G}_\mathsf{t}, \mathsf{m})$ holds, if G_t is $\mathsf{XMM}(\mathcal{M})$-terminal, then $\mathsf{Behavior}(\mathsf{G}_\mathsf{t}) = \mathsf{Behavior}(\mathsf{G}_\mathsf{s})$.

Then given a derivation of the target graph in the $\mathsf{XMM}(\mathcal{M})$ operational semantics $\mathsf{G}_\mathsf{t}^{\mathsf{init}} \longrightarrow^* \mathsf{G}_\mathsf{t}$ we reconstruct the derivation of the source graph in the $\mathsf{XMM}(\mathcal{M})$ semantics $\mathsf{G}_\mathsf{s}^{\mathsf{init}} \longrightarrow^* \mathsf{G}_\mathsf{s}$ using the lemmas stated above.

Our proofs are parametrized by the memory consistency model $\mathcal{M}$ used in $\mathsf{XMM}(\mathcal{M})$ operational semantics. For each transformation, we provide preconditions on the model $\mathcal{M}$ for the transformation to be sound. In total, we define *five memory consistency model properties*, required for all the considered transformations to be sound: *monotonicity*, *maximal read extensibility*, *maximal write extensibility*, *same-read extensibility*, *overwrite extensibility*. Intuitively, these properties have the following meaning.

- **Monotonicity** tells that any subgraph of a consistent execution graph has to remain consistent.
- **Maximal read extensibility** implies that a consistent execution can be extended with a new rpo-maximal read event, reading from a stable write (via srf relation), and remain consistent.
- **Maximal write extensibility** assures that a consistent execution graph can be extended with a new rpo-maximal and mo-maximal write event, and remain consistent.
- **Same-read extensibility** means that a consistent execution graph can be extended with a new read event, placed in program order po just after some existing read or write event, and as long as it reads-from this previous event (in case of write), or same write this read reads-from (in case of read), the execution remains consistent.
- **Overwrite extensibility** means that a consistent execution graph can be extended with a new write event, placed in program order po and in modification order mo just before some existing write event, and if no read reads from an added write event, the execution remains consistent.

Formal definitions of these properties are given in Appendix B.

In general, whenever the target graph performs a step $\mathsf{G}_\mathsf{t} \longrightarrow \mathsf{G}_\mathsf{t}'$, to simulate it, the source graph has to perform the same step $\mathsf{G}_\mathsf{s} \longrightarrow \mathsf{G}_\mathsf{s}'$, with the following exceptions.

- **Elimination.** In case of load-load and store-load elimination, whenever the first event a_t from the eliminated pair is added to the target graph: $\mathsf{G}_\mathsf{t} \xrightarrow[exec]{a_\mathsf{t}} \mathsf{G}_\mathsf{t}'$, we add to the source graph both events from the pair: $\mathsf{G_s} \xrightarrow[exec]{a_s}\xrightarrow[exec]{b_s} \mathsf{G'_s}$. The additional event b_s is a read event: in the case of load-load reordering, it reads from the same write as the read a_s, and in case of store-load reordering from the write a_s itself. In both cases, it can be shown that the added reads-from edge is stable $\mathsf{G_s}.\mathtt{srf}$ edge. Thus, the same-read extensibility property can be applied to derive the consistency of the resulting source graph $\mathsf{G'_s}$. In the case of store-store reordering, when the second event b_t is added to the target graph: $\mathsf{G_t} \xrightarrow[exec]{b_t} \mathsf{G'_t}$, we again add to the source graph both events from the pair: $\mathsf{G_s} \xrightarrow[exec]{a_s}\xrightarrow[exec]{b_s} \mathsf{G'_s}$, this time using the overwrite-extensibility property to derive the consistency of the resulting source graph $\mathsf{G'_s}$.

- **Reordering.** When a second event b_t from the reordered pair is added to the target graph: $\mathsf{G_t} \xrightarrow[exec]{b_t} \mathsf{G'_t}$, along it we add an intermediate event a'_s to the source: $\mathsf{G_s} \xrightarrow[exec]{a'_s}\xrightarrow[exec]{b_s} \mathsf{G'_s}$. When a'_s is a read event, we set it to read-from a stable write via $\mathsf{G'_s}.\mathtt{srf}$ relation, and when it is a write event, we put it at a maximal $\mathsf{G'_s}.\mathsf{mo}$ position. We then apply maximal read extensibility or maximal write extensibility property respectively to show the consistency of $\mathsf{G'_s}$. Later on, when the first event a_t from the reordered pair is added to the target graph: $\mathsf{G_t} \xrightarrow[exec]{a_t} \mathsf{G'_t}$, we perform a re-execution in the source graph: $\mathsf{G_s} \xrightarrow[re-exec]{C} \mathsf{G'_s}$, by committing all events except a'_s and b_s, and replacing a'_s with a_s during re-execution.
- **Sequentialization.** In the case of thread sequentialization, no special actions are required, as the source graph can mimic all types of steps taken by the target graph. During the construction, the source graph always has a smaller set of program order po, and therefore happens-before hb edges. Thus, the monotonicity property can be applied to prove the consistency of $\mathsf{G'_s}$.

When the target execution graph performs a re-execution step: $\mathsf{G_t} \xrightarrow[re-exec]{C_t} \mathsf{G'_t}$, we need to prove that the source graph can simulate this step $\mathsf{G_s} \xrightarrow[re-exec]{C_s} \mathsf{G'_s}$. As was shown above, the source graph almost always has the same set of events and the same relations as the target, except that in case of elimination and reordering, the source graph may also contain one additional event. Also, in case of reordering, the program order edge between a pair of recordable events is swapped. To show that nonetheless, it is still possible to simulate the re-execution process in the source, we utilize the Lemmas 1 and 2.

When the target graph performs re-execution $\mathsf{G_t} \xrightarrow[re-exec]{C_t} \mathsf{G'_t}$ with the given $\preceq_{\mathsf{tid_t}}$ thread ordering, we generally select the same relation for the source: $\preceq_{\mathsf{tid_s}} \triangleq \preceq_{\mathsf{tid_t}}$. The only exception is sequentialization: in this case, we additionally order a pair of thread ids $\langle t_i, t_j\rangle$ in the $\preceq_{\mathsf{tid_s}}$ relation to account for the sequentialized threads t_i and t_j. It can be shown that as long as $\mathit{ThreadOrderedUEvents}(\mathsf{G'_t}, C_t, \preceq_{\mathsf{tid_t}})$ holds, the predicate $\mathit{ThreadOrderedUEvents}(\mathsf{G'_s}, C_s, \preceq_{\mathsf{tid_s}})$ also holds. Indeed, $\mathsf{G'_s}$ may contain only one additional uncommitted read event in case of elimination or reordering, and this read event always reads-from a stable write event via $\mathsf{G'_s}.\mathtt{srf}$ relation. This added read event cannot participate in any new $\mathsf{po} \cup \mathsf{rf}; [\mathcal{U}]$ cycle. Therefore, Lemmas 1 and 2 can be applied to reconstruct re-execution steps for the source graph.

A more detailed proof can be found in Appendix E.

4 XMM Model Checking

In this section, we discuss the model checker XMC we have developed for the XMM semantics. Our model checker is based on the GenMC model checker [Kokologiannakis and Vafeiadis 2021]. Given a program, GenMC explores all its RC20-consistent executions (that is, all C20-consistent executions without porf cycles). To construct these executions, GenMC relies on a procedure resembling the (Execute) rule of the XMM semantics: at each step, it adds a single porf-maximal event and checks that the new execution graph remains C20-consistent. Thus, to generate porf-cyclic executions, we need to extend GenMC with a procedure following the (Re-Execute) rule.

However, in its general form, the (Re-Execute) rule exhibits significant non-determinism: it can be applied at any step with arbitrarily chosen sets of determined and committed events. To tame this non-determinism, we have developed a strategy that implements a restricted version of the (Re-Execute) rule, making the model checking tractable in practice. As we demonstrate empirically in §5, this restricted form of re-execution approximates the full XMM semantics surprisingly well, and the difference typically can be observed only on synthetic tests.

We restrict the re-execution capabilities in the following aspects.

- Instead of launching re-execution arbitrarily at any point, we only do it when a *load buffering race* is discovered, following the approach proposed by Moiseenko et al. [2022].

  Definition 15. A pair of read and write accesses $\langle r, w\rangle$ is in LBRace if they are in a read-write race, there is a $(\mathsf{po} \cup \mathsf{rf})$-path from r to w, and r does not read from w:

  $$\mathsf{LBRace} \triangleq \mathsf{Race} \cap [\mathsf{R}]; \mathsf{porf}; [\mathsf{W}] \setminus \mathsf{rf}^{-1}$$

- Instead of considering all possible combinations of the determined and committed event sets, we fix their choice based on the detected load buffering race $\langle r, w\rangle$. We set the determined events $\mathcal{D}$ to be all the events except those from the program-order suffix of r. Also, we set the committed events C to contain all write events from the same suffix that are read externally.

  $$\mathcal{D} \triangleq \mathsf{G.E} \setminus \mathsf{codom}[r]; \mathsf{G.po}^{?} \quad \text{and} \quad C \triangleq \mathcal{D} \cup \mathsf{codom}([r]; \mathsf{G.po}^{?}) \cap \mathsf{dom}(\mathsf{G.rf} \setminus \mathsf{G.po})$$

  As such, we always re-execute only a single thread — the one where the racy read r occurred, and therefore the choice of the thread ordering $\preceq_{\mathsf{tid}}$ degenerates to a trivial one.

4.1 XMC Model Checking Algorithm

The XMC model checker relies on the Visit(P, G) procedure provided by GenMC [Kokologiannakis et al. 2022]. Given a program P and its partially constructed graph G, this procedure generates all terminal execution graphs of P that can be constructed by extending G with new events. Internally, this procedure at each step inserts a new porf maximal event into a graph, calling Visit recursively to consider all possible further extensions of the graph.

The XMC re-execution algorithm is given in Algorithm 1. Overall, we follow monadic 'do' notation in the algorithms where $a \leftarrow A$ denotes that a is non-deterministically selected from a set A and $a := b$ denotes that b is assigned to a.

Algorithm 1: XMC Model checking re-execute algorithm.

```
Procedure reeXecute($\mathbb{P}$, G)
    if $\neg$isCons(G) $\vee$ $\neg$embeddedSubGraph(G) $\vee$ duplicate(G) then
        return;
    output (G);
    $\langle r, w\rangle \leftarrow$ G.LBRace ;
    if $r = \bot$ then
        return ;
    visitLBRaces ($\mathbb{P}$, G, $r$) ;                                // Phase I
    visitROutCycle ($\mathbb{P}$, G, $r$) ;                              // Phase II
    visitRORacyR ($\mathbb{P}$, G, $r$) ;                                // Phase III
```

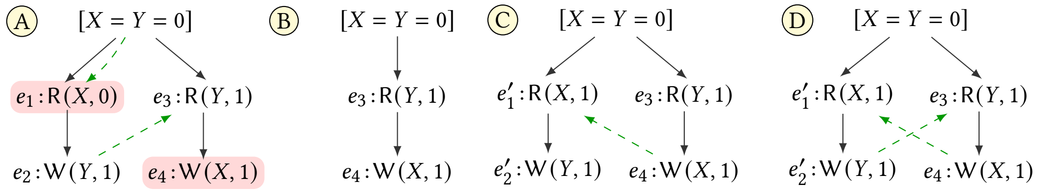


Fig. 7. XMC steps for re-execution to create $a = b = 1$ outcome.

We have modified the GenMC Visit procedure so that for each generated terminal execution graph G it calls a new routine reeXecute(P, G). This procedure performs some additional checks before outputting the graph. Next, Line 5 non-deterministically picks a LBRacy read r. If G has no such LBRace then we do not attempt to construct any porf cycle anymore and hence we return in Line 7. Otherwise, we perform re-executions to create new porf cyclic graphs. The re-execution is performed in three phases:

(Phase I) Re-execute load buffering race (Line 8).
(Phase II) Revisit reads outside cycle (Line 9).
(Phase III) Revisit reads in the re-executed thread (Line 10).

(Phase I) Re-execute load buffering race. In this phase, we choose the sets of committed and determined events based on a given read r in LBRace. Let $G_r \subseteq G$ be a restricted graph without r and its po-suffix events. We select the events in G_r as determined events. Note that, G_r contains a set of reads $R_\bot$ without any incoming G_r.rf edges. The set of committed events consists of the G_r events and the writes in G from which the reads events $R_\bot$ read-from.

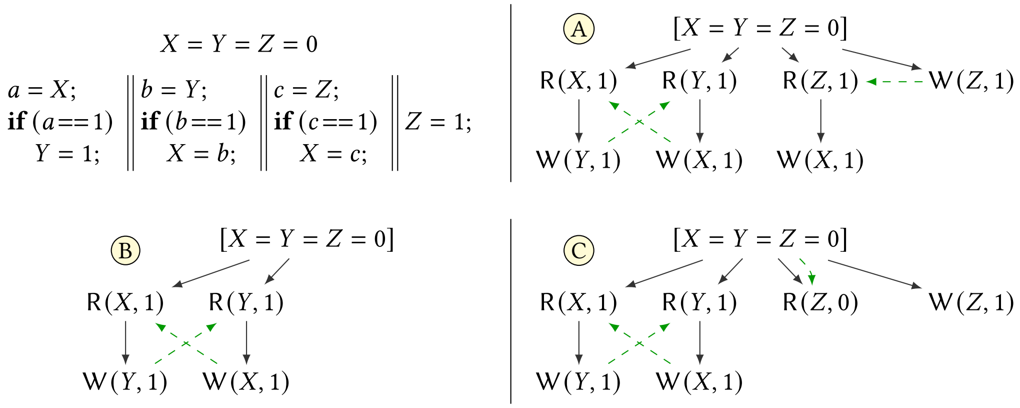
Example. Consider the RC20 execution of the LB program in Ⓐ in Figure 7. We detect a LBRace between events e_1 and e_4 that are highlighted in red. The execution graph G_r is in Ⓑ, and the $\mathcal{D} \triangleq \{e_3, e_4\}$ and $C \triangleq \{e'_2, e_3, e_4\}$. Note that $R_\bot = \{e_3\}$. Next, we re-execute the thread 1 that results in $e'_1 : R(X, 1)$ reading from event e_4, followed by $e'_2 : W(Y, 1)$ in Ⓒ. We also create an rf edge from e'_2 to e_3 in Ⓓ. The resulting execution in ④ is XC20 consistent.

Algorithm. The algorithm for phase I is defined in Algorithm 2 in two procedures visitLBRaces and matchRfEdges along with other helper functions.

- The visitLBRaces takes program $\mathbb{P}$, an execution graph G, and a read r that is in LBRace. In Line 2 we use r to compute the set of determined events $\mathcal{D}$, which contains all the events of the graph G except r and its po-prefix. Next, the visit function returns a set of graphs constructed from G from which we select one graph $G_\bot$ in Line 3. The $G_\bot$ graph may contain a set of reads without incoming rf edges. We call matchRfEdges to create the new rf edges for these reads and construct the rf-complete execution graph G' in Line 4. Finally, we recursively call reeXecute on G' to explore further re-executions in Line 5.
- The matchRfEdges procedure identifies the writes for the $R_\bot$ events. It takes as arguments two graphs: the original graph G_o and a derived graph $G_\bot$ containing $R_\bot$ events. In Line 7, we non-deterministically pick a read r that misses its reads-from edge from $G_\bot$. We check if no such r exists in Line 8 and return graph $G_\bot$ in that case. Otherwise, in Line 10, we find the original reads-from write w in the graph G_o. In Line 11, we non-deterministically pick a write w' from $G_\bot$ such that the thread id, location, and value of w' match those of w. Next, in Line 12 we use auxiliary function changeRF to create a new rf edge $\langle w', r \rangle$ in $G_\bot$ and call matchRfEdges recursively to search for the remaining reads-from writes.

Algorithm 2: XMC Model checking cycle creation (Phase I).

```
1  Procedure visitLBRaces($\mathbb{P}$, G, $r$)
2      $\mathcal{D}$ := G.E \ codom([{$r$}]; G.po$^?$);
3      $G_\bot \leftarrow$ visit($\mathbb{P}$, G$|_\mathcal{D}$);
4      $G' \leftarrow$ matchRfEdges($G_\bot$, G);
5      reeXecute ($\mathbb{P}$, G$'$);
6  Procedure matchRfEdges($G_\bot$, $G_o$)
7      $r_\bot \leftarrow G_\bot$.R \ codom($G_\bot$.rf);
8      if $r_\bot = \bot$ then
9          return $G_\bot$;
10     $w \leftarrow$ dom($G_o$.rf; [$r_\bot$]);
11     $w' \leftarrow G_\bot.\mathrm{E}_{\mathrm{tid}(w)} \cap G_\bot.\mathrm{W}_{\mathrm{loc}(w)} \cap G_\bot.\mathrm{W}_{\mathrm{val}(w)}$;
12     matchRfEdges($G_\bot$.changeRF($w', r_\bot$), $G_o$);
```



```
X = Y = Z = 0

a = X;          || b = Y;          || c = Z;          || Z = 1;
if (a==1)       || if (b==1)       || if (c==1)       ||
  Y = 1;        ||   X = b;        ||   X = c;        ||
```

Fig. 8. Outcome $r_1 = r_2 = 1, r_3 = 0$ is obtained in phase II.

4.1.1 **(Phase II) Revisit reads outside cycle.** After obtaining a porf cyclic execution, it is possible that some reads that are not in the cycle or its porf-prefix can be revisited to read from some happens-before hb preceding writes. We achieve this by restricting the graph, keeping all events that are part of the given porf cycle and its porf-prefix, and removing everything else. To derive new graphs, we re-execute starting from this restricted graph.

Example. Consider the Java causality test 10 [Litmus [n. d.]] in Figure 8. In this program, the outcome $r_1 = r_2 = 1, r_3 = 0$ can be obtained by sequentialization transformation [Chakraborty and Vafeiadis 2019] and the behavior is allowed by XMM. To obtain the execution, we first create a porf cycle in phase I where the execution has $r_1 = r_2 = r_3 = 1$ as shown in Ⓐ. At this step, we execute phase II considering that R$(X, 1)$ is in LBRace. The determined events are in porf-prefix of R$(X, 1)$ events and we remove the rest of the events from the graph which is shown in Ⓑ. Next, we re-execute the execution and obtain the completed graph in Ⓒ where the read of Z reads the initial value and results in $r_1 = r_2 = 1, r_3 = 0$.

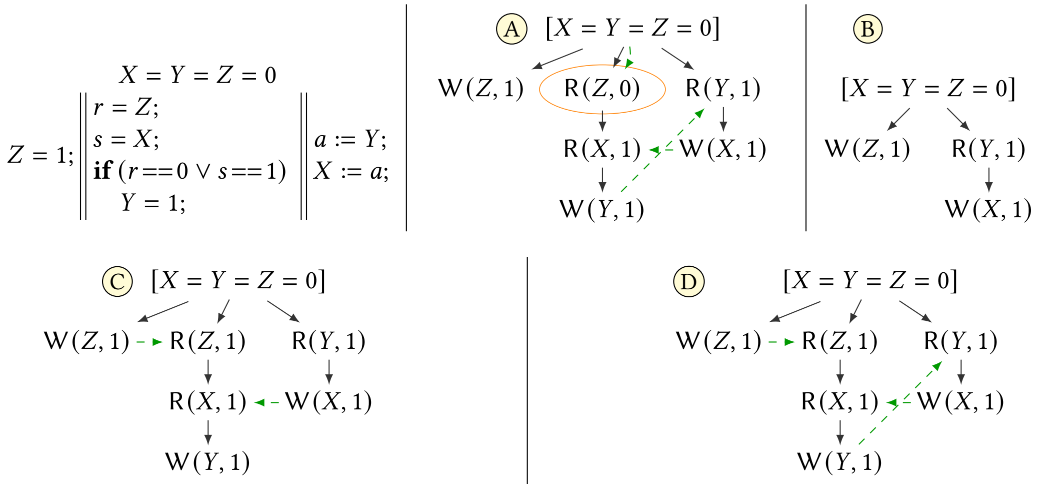
Algorithm. The algorithm for phase II is defined in the procedure visitROutCycle. In Line 2 we restrict the graph to the porf prefix of the read r detected to be in LBRace. Note that this restricted graph G$|_\mathcal{D}$ is complete, that is, all reads have incoming rf edges. Then, we perform re-execution by calling the visit in Line 3. It returns a set of complete graphs from which we select G$'$. Next, we call reeXecute in Line 4, which outputs G$'$ and looks for new LB races in it.

(Phase III) Revisit reads in the re-executed thread. After a cycle is created in phase I, we revisit read events in the thread of the lb-racy read r to account for new potential rf edges.

Algorithm 3: XMC (Phase II) for revisiting outside porf cycle and (phase III) for revisiting threads with LBRace reads.

```
1  Procedure visitROutCycle($\mathbb{P}$, G, $r$)
2      $\mathcal{D}$ := dom(G.porf$^?$; [{$r$}]);
3      G$'$ ← visit($\mathbb{P}$, G$|_{\mathcal{D}}$);
4      reeXecute ($\mathbb{P}$, G$'$);

5  Procedure visitRORacyR($\mathbb{P}$, G, $r$)
6      $r'$ ← dom(G.po; [{$r$}]) ∩ G.R;
7      $\mathcal{D}$ := G.E \ codom([$r'$]; G.po$^?$) ;
8      G$'$ ← visit($\mathbb{P}$, G$|_{\mathcal{D}}$);
9      G$''$ ← matchRfEdges(G$'$, G);
10     reeXecute($\mathbb{P}$, G$''$);
```



```
            X = Y = Z = 0
       ║ r = Z;               ║ a := Y;
Z = 1; ║ s = X;               ║ X := a;
       ║ if (r==0 ∨ s==1)     ║
       ║   Y = 1;             ║
```

Fig. 9. Revisit reads in phase III.

Example. Consider the example in Figure 9 from Moiseenko et al. [2022] where $r = s = a = 1$ is a desired outcome. Suppose phase I constructs the execution shown in Ⓐ where R$(X, 1)$ was identified to be in LBRace. In that case, R$(Z, 0)$ is selected to be revisited. As shown in Ⓑ, we remove R$(Z, 0)$ event along with its po-suffix events R$(X, 1)$ and W$(Y, 1)$, it results in R$(Y, 1)$ event without incoming rf edge. The re-execution constructs the events of the second thread in Ⓒ where R$(Z, 1)$ is created. We also create the R$(X, 1)$ event that satisfy the **if**-condition and consequently create the W$(Y, 1)$ event. Finally, in Ⓓ we create an rf edge from W$(Y, 1)$ to R$(Y, 1)$ arriving at the desired outcome.

Algorithm. The algorithm for phase III is defined in the procedure visitRORacyR in Algorithm 3. In Line 6, given a read event r which was in LBRace, we identify the set of reads from its po-prefix. We select one of these reads non-deterministically, say r'. We restrict the graph to $\mathcal{D}$ by removing r' along with its po-suffix in Line 7. Next, we re-execute in Line 3 by visit that generate a set of graphs from which we non-deterministically select a graph, say G$'$. The execution graph G$'$ may have read events without incoming rf edges. In Line 9 we complete the graph by creating the rf-edges. Finally, we call reeXecute in Line 10 that outputs G$''$ and searches for new load-buffering races to continue exploration.

Additional Checks. Before outputting the execution graph, the reeXecute procedure in Algorithm 1 also performs the following checks in Line 2.

- **Consistency.** It is imperative to check if the generated execution graph is consistent. Note that normally, the Visit procedure in GenMC would also check the consistency of each intermediate execution graph at each step after a new event is added. But, during the re-execution process, XMC postpones the consistency checks and performs only the final check after all $\mathsf{R}_\bot$ reads are matched, discarding the inconsistent graphs. Otherwise, the consistency checking routine of GenMC may fail, as it assumes that the execution graph is *RfComplete*.

- **Duplicate executions.** The GenMC algorithm ensures that it never creates any duplicate execution [Kokologiannakis et al. 2022], relying on the porf acyclicity of the RC20 memory model. However, as was shown in [Moiseenko et al. 2022], the same strategy cannot be repurposed for a memory model permitting porf cycles. As such, we memorize the graphs to check for duplicates once the graph is constructed and completed.
- **Embedded subgraph.** By definition, the Visit procedure of GenMC ensures all the conditions of the committed subgraph embedding *CommitEmbedded* given in Definition 7 hold, except the ones involving rpo and mo relations. Thus, we additionally check that rpo and mo ordering on the committed events is preserved between the original and the constructed graphs.
- **Thread-ordered reads.** Note that because we always re-execute only a single thread, the *ThreadOrderedUEvents* predicate given in Definition 8 is satisfied trivially.

Algorithm Analysis. The XMC algorithm judiciously chooses the determined and committed events $\mathcal{D}$ and C instead of exploring all possible subsets, as allowed by XMM. As such, we show that each re-execution step that XMC performs is a special case of the general (Re-Execute) rule of XMM, and thus XMC generates only XMM consistent execution graphs.

Theorem 4.1 (Soundness Theorem). XMC *is sound. The proof can be found in Appendix F.*

Completeness and optimality. The XMC algorithm does not provide a completeness property as it does not explore all possible subsets of the committed and determined events. The XMC algorithm is also not optimal as it may create duplicate execution graphs and discard them.

5 Experimental Evaluation

We evaluated the XMC model checker on a set of well-known litmus tests and benchmarks taken from the literature [Abdulla et al. 2017; Jagadeesan et al. 2020; Litmus [n. d.]; Moiseenko et al. 2022; Norris and Demsky 2013]. The details of these benchmarks are given in Appendix G.

The benchmarks were run in a Docker container on a macOS machine with a 2.9 GHz Intel Core i9 CPU and 32 GB of memory. The reported execution times were averaged over five runs.

We compare the performance of XMC to the following model checkers.

- GenMC$_\mathsf{X}$ [Kokologiannakis and Vafeiadis 2021] is the base version of the GenMC model checker on which XMC was developed. It supports the RC20 memory model and thus explores only porf acyclic executions.
- HMC [Kokologiannakis and Vafeiadis 2020] is a variant of the GenMC model checker which supports the IMM model [Podkopaev et al. 2019] that provides an abstraction over the hardware concurrency models. It is capable of exploring some subset of porf cyclic executions which do not have cycles consisting of *syntactic dependencies* and reads-from edges.
- WMC [Moiseenko et al. 2022] is an extension of the GenMC model checker which supports a variant of Weakestmo memory model — a multi-execution event-structure based memory model [Chakraborty and Vafeiadis 2019]. Similarly to XMC, this model checker explores the executions with porf cycles based on detected load buffering races. However, compared to XMC, it supports only a narrowed subset of porf cyclic executions, since it also relies on a stronger memory model property of *certification locality*. This property is incompatible with the sequentialization transformation.

- GenMC$_{\text{W}}$ is an older version of the GenMC model checker on which WMC was developed [Moiseenko et al. 2022]. Since GenMC is continually being developed, the two versions of the tool are actually differing significantly. These differences may affect the execution time of the tools. Thus, for a fairer comparison, we used both baseline versions of GenMC in our experiments.

Given the setup and benchmarks, we explore the following research questions (RQ).

- RQ1: How does XMC perform compared to the other model checkers in exploring all XMM executions optimally for the litmus tests? (§5.1)
- RQ2: How effective is XMC in verifying real-world lock-free data structures compared to other model checkers? (§5.2)

5.1 Evaluating XMC on Litmus Tests (RQ1)

We have experimented on 73 synthetic litmus tests with load buffering races. These tests comprise both established tests from the literature and new tests that we have introduced. We compared the XMC behavior on these tests with the other model checkers. We do not report the execution times of the model checkers for these small tests as they are negligible, we are only interested in the number of execution graphs they explore.

Table 1. XMC explores more executions for the litmus tests compared to the other model checkers.

Litmus tests	GenMC	HMC	WMC	XMC
RMulMatch	15	18	20	21
LBWDep	2	2	2	3
LB+equals	9	9	9	10
java-test5	20	20	24	28
java-test9a	10	10	10	12
java-test10	5	5	5	8
java-test19	14	14	17	20
java-test20	14	14	17	20
LB+seq-src	14	14	17	20
LB-coh-RR-cf	24	24	24	36
LB+coh-cyc	5	5	5	6
LB+coh-cyc-Wd	10	10	10	12

Table 1 shows the numbers of executions for 12 tests where XMC explores more executions than WMC— its closest competitor. Among all the model checkers, GenMC (both its versions GenMC$_{\text{W}}$ and GenMC$_{\text{X}}$) only explore porf-acyclic executions. As expected, they report the same number of executions, and thus are shown in one column. HMC explores more execution than GenMC only in *RMulMatch*. WMC follows a weaker semantics than HMC and consequently explores additional executions in 5 tests: *RMulMatch*, *java-test19*, *java-test20*, *java-test5*, and *LB+seq-src*.

XMC explores more execution than all these model checkers including WMC in all 12 tests. In the case of *LB+coh+RR+cf* test, this is because WMC enforces a global event-structure-level constraint on the mo order, while XMC does not have this constraint. XMC missed XMM consistent executions only in four cases: *LB-DRF*, *LB-DRF-SLI*, *LB-DRF-LL*, *LB+coh-cyc*, and *LB+porf-suffix*.

5.2 Evaluating XMC on data-structure benchmarks (RQ2)

Table 2 reports the numbers of explored executions and verification times by the model checkers for all concurrent data structure benchmarks. As shown, all these benchmarks contain load buffering races. The numbers of explored executions remain the same in all benchmarks except *fc-async*, *linuxrwlk*, *dq*, *chase-lev*. In *fc-async*, and *linuxrwlk* benchmarks the GenMC$_{\text{W}}$ and WMC times out with a threshold of 60 seconds. In *chase-lev* the numbers of explored executions in GenMC$_{\text{X}}$ and GenMC$_{\text{W}}$ differ. These variations are due to the bug fixes and improvements added to a newer version GenMC$_{\text{X}}$. These improvements are also reflected in HMC. In the *dq* benchmark XMC explores 141 additional executions, whereas WMC explores no additional execution.

XMC incurs more time in exploring the additional executions compared to that of WMC, particularly with the increase of the load buffering races. We attribute this to a difference between two basic versions of GenMC: GenMC$_{\text{X}}$ and GenMC$_{\text{W}}$. They employ different strategies during graph

Table 2. Number of executions explored and time (T) on data structure benchmarks. XMC explores more executions than other tools in the *chase-lev* and *dq* tests. The threshold for timeout is 60 seconds.

Testcases	lbRace (#)	HMC		GenMC$_W$		WMC		GenMC$_X$		XMC	
		exec (#)	T (s)	exec (#)	T (s)	exec (#)	T (s)	exec (#)	T (s)	exec (#)	T (s)
mutex	30	12	0.05	12	0.02	12	0.02	12	0.03	12	0.04
fc-async	32	24	0.36	🕒	🕒	🕒	🕒	24	0.33	24	0.34
twalock	288	96	0.34	96	0.03	96	0.03	96	0.34	96	0.41
linuxrwlk	384	216	0.40	🕒	🕒	🕒	🕒	216	0.23	216	0.81
stc	1515	183	0.15	183	0.05	183	0.07	183	0.14	183	0.30
dq	1616	1924	0.28	1802	0.14	1802	0.15	1924	0.18	2065	2.79
buf-ring	2172	1218	3.92	1218	1.54	1218	1.59	1218	1.72	1218	2.28
chase-lev	6530	3639	1.91	3809	0.58	3975	0.66	3639	0.74	3821	1.95
Tktlock	10800	720	2.01	720	3.40	720	3.65	720	0.34	720	48.43
mpmcQ	24240	15752	9.26	15752	4.00	15752	4.31	15752	4.81	15752	12.64

exploration. When encountering a so-called *backward revisit*, that is a revisit of a previously added read event r by a newly added write event w, GenMC$_X$ makes a full copy of a graph for future exploration, while GenMC$_W$ copies only a part of the graph. Since every detected load buffering race leads to a backward revisit, a lot of graph copies are created for future explorations.

However, as can be seen from the table, most of these explorations are fruitless as they do not lead to new execution graphs. As an optimization, we could try to employ some heuristics that would help the algorithm to discard these infeasible explorations earlier, before a graph copy is created. We leave this for future work.

6 Related Work

Defining relaxed memory concurrency semantics for programming languages that address the associated challenges is widely explored [Batty et al. 2015]. Adve and Hill [1990] defined the data-race-free-0 (DRF0) model that differentiates between the *data* and *synchronization* variables. In the DRF0 model, races on synchronization variables are allowed and races on data variables result in undefined behavior. The DRF0 model provided the foundation for the concurrency model for the Java and C/C++ programming languages [Boehm and Adve 2008; Gosling et al. 1996]. Pugh [1999] identified that several compiler optimizations are unsound in the Java Memory Model specification [Gosling et al. 1996]. The Java memory model was revised by Manson et al. [2005b] to provide DRF-SC guarantee and support reordering transformations. In this model, a set of events is committed at each step and alternative well-formed executions are identified. However, Aspinall and Ševčík [2007] discovered flaws in this semantics, some of which were later fixed by Ševčík and Aspinall [2008]. Our approach is inspired by the concepts of committed events and alternative well-formed execution in Manson et al. [2005b], however, our commit-and-re-execute strategy significantly differs to support the desired properties.

The initial concurrency model for C/C++ [ISO/IEC 14882 2011; ISO/IEC 9899 2011], well known as C11, was proposed by Boehm and Adve [2008] and was formalized by [Batty et al. 2011]. The model was later found to have several limitations [Batty et al. 2013; Vafeiadis et al. 2015]. Since then, several models have been proposed to address different limitations of C/C++ concurrency – both in per-execution [Batty et al. 2016; Lahav et al. 2017; Margalit and Lahav 2021; Vafeiadis et al. 2015] and in multi-execution manner [Chakraborty and Vafeiadis 2019; Kang et al. 2017; Lee et al. 2020; Pichon-Pharabod and Sewell 2016; Wright et al. 2023].

Among the multi-execution models, promising semantics (PS) [Kang et al. 2017; Lee et al. 2020] is defined operationally. In PS, each write operation creates a message with a timestamp extending various views of the thread. A read operation reads from a message and extends the thread views with the message views. In addition, promising semantics allow a thread to *promise* a write operation (resulting in a message) which can be later fulfilled if the thread has a thread-local certificate. PS differentiates between the LBD, LB, LBfd programs, satisfies DRF properties, and supports certain desired compiler optimizations. However, PS does not handle SC memory accesses and lacks support for sequentialization. Pichon-Pharabod and Sewell [2016] defines the semantic model based on event structure, baking in various desired optimizations. However, the model does not provide DRF guarantees. Jeffrey and Riely [2016] provides an event structure-based model, primarily for a subset of Java accesses. The model satisfies DRF guarantees but does not support read-read reordering and provides weaker coherence properties. Chakraborty and Vafeiadis [2019] proposed an axiomatic-operational model for event structure construction followed by execution extraction. As mentioned in the introduction, it lacks support for sequentialization. The models in Jeffrey et al. [2022b]; Paviotti et al. [2020] provide certain compositional properties but do not support all the optimizations. Given the constructs of PS and event structure-based models, it is unclear if they can be extended to support sequentialization along with other properties.

Besides compiler optimization, efficient mapping schemes from C/C++ concurrency primitives to hardware instructions are extensively studied for architectures including x86-TSO, ARM, Power, RISC-V [Batty et al. 2011; Chakraborty and Vafeiadis 2019; Kang et al. 2017; Lahav et al. 2017; Lee et al. 2020; Podkopaev et al. 2019; Sarkar et al. 2012, 2011]. Podkopaev et al. [2019] proposed IMM as an abstraction on these architectures to develop mechanized proofs of the mapping correctness from various programming language semantic models to these architectures.

Semantic models provide the guiding principles to developing automated reasoning techniques by model checking [Abdulla et al. 2017; Kokologiannakis and Vafeiadis 2021; Norris and Demsky 2013], dynamic analysis, and testing [Gao et al. 2023; Luo and Demsky 2021; Tunç et al. 2023]. These analyses follow the acyclicity of porf relations in the respective semantic models. On the other hand, developing analysis tools that allow porf cycles are more challenging [Moiseenko et al. 2022; Pulte et al. 2019]. Pulte et al. [2019] developed a model checker for a fragment of PS and Moiseenko et al. [2022] developed WMC model checker following Weakestmo2 event structure based semantics [Moiseenko et al. 2020]. Compared to WMC, the XMM model checker provides soundness guarantees.

7 Conclusion & Future Work

We propose a concurrency semantic framework based on re-execution strategy that satisfies the desiderata of desired properties for concurrency semantics in programming languages. While we focus on C/C++ concurrency primitives and consistency properties, our model is parameterized and can be used to model different concurrency semantics. Our approach checks consistency per-execution and avoids the complexities of multi-execution models that reason about multiple executions together. Our model is executable. To demonstrate this, we have developed a sound model checker to explore the executions with porf cycles of various benchmark programs.

In the future, we want to develop different re-execution strategies for encoding other concurrency models. Another goal is to improve the model checker to obtain completeness and optimality.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback.

References

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS 2015 (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

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 (dec 2017), 789–818. https://doi.org/10.1007/s00236-016-0275-0

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering&Mdash;a New Definition. In *ISCA '90*. 2–14. https://doi.org/10.1145/325164.325100

David Aspinall and Jaroslav Ševčík. 2007. Formalising Java's Data Race Free Guarantee. In *Theorem Proving in Higher Order Logics*, Klaus Schneider and Jens Brandt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–37.

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

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *POPL '16*. ACM, 634–648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP'15*. 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, 55–66. https://doi.org/10.1145/1926385.1926394

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *PLDI'08*. https://doi.org/10.1145/1375581.1375591

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

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI 2016*. 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *ASPLOS, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3575693.3575729

James Gosling, Bill Joy, and Guy Steele. 1996. The Java Language Specification. *Addison Wesley* (1996).

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

ISO/IEC 9899. 2011. Programming Language C. (2011).

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

Alan Jeffrey and James Riely. 2016. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *LICS'16*. ACM/IEEE. https://doi.org/10.1145/2933575.2934536

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022a. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–30.

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022b. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498716

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL'17*. ACM.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–28.

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *ASPLOS 2020 (ASPLOS '20)*. 1157–1171. https://doi.org/10.1145/3373376.3378480

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, Cham, 427–440.

Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory models fair. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. 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

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

3385412.3386010

Litmus. [n. d.]. Causality Test Cases. http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html. ([n. d.]).

Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In *ASPLOS 2021*. 630–646. https://doi.org/10.1145/3445814.3446711

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005a. The Java Memory Model. In *POPL'05*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005b. The Java Memory Model. In *POPL '05*. ACM.

Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a C11-style memory model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (jan 2021). https://doi.org/10.1145/3434285

Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. 2022. Model checking for a multi-execution memory model. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 152 (oct 2022), 28 pages. https://doi.org/10.1145/3563315

Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. 2025. Relaxed Memory Concurrency Re-executed (Technical Appendix). (2025). Available at https://eupp.github.io/papers/popl2025-xmm-apdx.pdf.

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2020. Reconciling Event Structures with Modern Multiprocessors. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.5

Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *OOPSLA'13*.

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Vol. 12075. 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

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, 622–633. https://doi.org/10.1145/2676726.2676995

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

William Pugh. 1999. Fixing the Java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA '99)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/304065.304106

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *PLDI 2019 (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI'12*. ACM, 311–322. https://doi.org/10.1145/2254064.2254102

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *PLDI '11*. 175–186.

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–51.

Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 137 (jun 2023). https://doi.org/10.1145/3591251

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'15*. ACM, 209–220. https://doi.org/10.1145/2676726.2676995

Daniel Wright, Sadegh Dalvandi, Mark Batty, and Brijesh Dongol. 2023. Mechanised Operational Reasoning for C11 Programs with Relaxed Dependencies. *Form. Asp. Comput.* 35, 2, Article 10 (jun 2023). https://doi.org/10.1145/3580285