# Optimal Reads-From Consistency Checking for C11-Style Memory Models

HÜNKAR CAN TUNÇ, Aarhus University, Denmark
PAROSH AZIZ ABDULLA, Uppsala University, Sweden
SOHAM CHAKRABORTY, TU Delft, Netherlands
SHANKARANARAYANAN KRISHNA, IIT Bombay, India
UMANG MATHUR, National University of Singapore, Singapore
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

Over the years, several memory models have been proposed to capture the subtle concurrency semantics of C/C++. One of the most fundamental problems associated with a memory model $\mathcal{M}$ is *consistency checking*: given an execution X, is X consistent with $\mathcal{M}$? This problem lies at the heart of numerous applications, including specification testing and litmus tests, stateless model checking, and dynamic analyses. As such, it has been explored extensively and its complexity is well-understood for traditional models like SC and TSO. However, less is known for the numerous model variants of C/C++, for which the problem becomes challenging due to the intricacies of their concurrency primitives. In this work we study the problem of consistency checking for popular variants of the C11 memory model, in particular, the RC20 model, its release-acquire (RA) fragment, the strong and weak variants of RA (SRA and WRA), as well as the Relaxed fragment of RC20.

Motivated by applications in testing and model checking, we focus on *reads-from* consistency checking. The input is an execution X specifying a set of events, their program order and their reads-from relation, and the task is to decide the existence of a modification order on the writes of X that makes X consistent in a memory model. We draw a rich complexity landscape for this problem; our results include (i) nearly-linear-time algorithms for certain variants, which improve over prior results, (ii) fine-grained optimality results, as well as (iii) matching upper and lower bounds ($\mathcal{NP}$-hardness) for other variants. To our knowledge, this is the first work to characterize the complexity of consistency checking for C11 memory models. We have implemented our algorithms inside the TruSt model checker and the C11Tester testing tool. Experiments on standard benchmarks show that our new algorithms improve consistency checking, often by a significant margin.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis*.

Additional Key Words and Phrases: concurrency, weak memory models, complexity

Authors' addresses: Hünkar Can Tunç, Aarhus University, Denmark, tunc@cs.au.dk; Parosh Aziz Abdulla, Uppsala University, Sweden, parosh@it.uu.se; Soham Chakraborty, TU Delft, Netherlands, s.s.chakraborty@tudelft.nl; Shankaranarayanan Krishna, IIT Bombay, India, krishnas@cse.iitb.ac.in; Umang Mathur, National University of Singapore, Singapore, umathur@comp.nus.edu.sg; Andreas Pavlogiannis, Aarhus University, Denmark, pavlogiannis@cs.au.dk.


2475-1421/2023/6-ART137
https://doi.org/10.1145/3591251

**ACM Reference Format:**

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 (June 2023), 25 pages. https://doi.org/10.1145/3591251

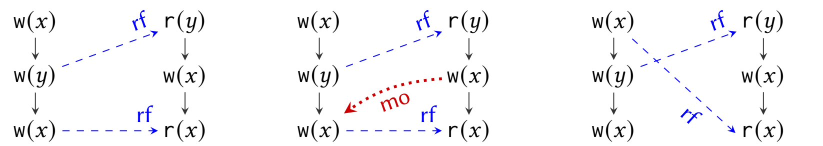
## 1 INTRODUCTION

Modern programming languages such as C/C++ [ISO/IEC 14882 2011; ISO/IEC 9899 2011] have introduced first-class platform-independent concurrency primitives to gain performance from weak memory architectures. The programming model is popularly known as C11 [Batty et al. 2011; Boehm and Adve 2008]. The formal semantics of C11 has been an active area of research [Batty et al. 2016, 2011; Chakraborty and Vafeiadis 2019; Lahav et al. 2017; Lee et al. 2020; Margalit and Lahav 2021; Vafeiadis et al. 2015] and other programming languages such as Java [Bender and Palsberg 2019] and Rust [Dang et al. 2019] have also adopted similar concurrency primitives.

One of the most fundamental computational problems associated with a memory model, particularly in testing and verification, is that of *consistency checking* [Furbach et al. 2015; Gibbons and Korach 1997; Kokologiannakis et al. 2023]. Here, one focuses on a fixed memory model $\mathcal{M}$, typically described using constraints or axioms. The input to the consistency problem pertaining to the memory model $\mathcal{M}$ is then a partial execution $\mathsf{X}$, typically described using a set of events $\mathsf{E}$ together with a set of relations on $\mathsf{E}$. The consistency problem then asks to determine if $\mathsf{X}$ is *consistent* with $\mathcal{M}$. Here, by *partial* execution, we mean that the set of relations is not fully described in the input, in which case the problem asks whether $\mathsf{X}$ can be extended to a complete execution that is consistent with $\mathcal{M}$. The focus of this paper is *reads-from* consistency checking; in the rest of the paper we refer to this simply as consistency checking.

The problem of consistency checking has numerous applications in both software and hardware verification. First, viewing memory models as contracts between the system designer and the software developers, consistency checking is a common approach to testing memory subsystems, cache-coherence protocols and compiler transformations against the desired contract [Chen et al. 2009; Manovit and Hangal 2006; Qadeer 2003; Wickerson et al. 2017; Windsor et al. 2022]. Second, since public documentations of memory architectures are typically not entirely formal, litmus tests can reveal or dismiss behaviors that are not covered in the documentation [Alglave 2010; Alglave et al. 2011, 2014]. Consistency checking for litmus tests makes testing more efficient (and thus also more scalable), by avoiding the enumeration of behaviors that are impossible under the given model. Third, in the area of model checking, (partial) executions typically serve the role of abstraction mechanisms. Consistency checking, thus, ensures that model checkers indeed explore valid system behavior. As such, it has been instrumental in guiding recent research in partial-order reduction techniques and stateless model checking of concurrent software [Abdulla et al. 2019, 2018; Agarwal et al. 2021; Bui et al. 2021; Chalupa et al. 2017; Chatterjee et al. 2019; Kokologiannakis et al. 2022; Norris and Demsky 2013]. Focusing on partial executions allows such algorithms to consider coarser equivalences such as the *reads-from* equivalence, allowing for more proactive state-space reductions and better performance as a result. These advances have also propelled the use of formal methods in the industry [Bornholt et al. 2021; Lerche 2020; Oberhauser et al. 2021]. Consistency checking of partial executions also forms the foundation of dynamic *predictive analyses* by characterizing the space of perturbations that can be applied to an observed execution in an attempt to expose a bug [Huang et al. 2014; Kalhauge and Palsberg 2018; Kini et al. 2017; Luo and Demsky 2021; Mathur et al. 2018, 2020, 2021; Pavlogiannis 2019].

$$
\begin{array}{l||l}
x := 1; & a := y; \\
y := 1; & x := 3; \\
x := 2; & b := x;
\end{array}
$$




(a) Program (b) RA-consistent execution. (c) RA-consistent mo. (d) RA-inconsistent execution.

Fig. 1. A program (a) and a partial execution $\overline{\mathsf{X}}$ specifying the writer $\mathsf{rf}^{-1}(\mathsf{r})$ of each read r (b). $\overline{\mathsf{X}}$ is RA-consistent, as witnessed by the modification order mo that abides to RA semantics (c). The partial execution in (d) is RA-inconsistent, as there is no modification order mo that abides to RA semantics.

The ubiquitous relevance of consistency checking has led to a systematic study of its computational complexity under various memory models. Under sequential consistency (SC), most variants of the problem were shown to be $\mathcal{NP}$-hard in the seminal work of Gibbons and Korach [Gibbons and Korach 1997]. Subsequently, more fine-grained investigations have characterized how input parameters such as the number of threads, memory locations, write accesses and communication topology affect the complexity of consistency checking [Abdulla et al. 2019; Agarwal et al. 2021; Chini and Saivasan 2020; Mathur et al. 2020]. As the consistency problems remain intractable under most common weak memory models (such as SPARC/X86-TSO, PSO, RMO, PRAM) [Furbach et al. 2015], parametric results have also been established for these models [Bui et al. 2021; Chini and Saivasan 2020]. Given its applications in analysis of concurrent programs, clever heuristics have been proposed to enhance the efficiency of checking consistency in practice [Zennou et al. 2019].

The C11 memory model provides the flexibility to derive different weak memory model paradigms based on different subsets and combinations of the concurrency primitives, their memory orders, and their respective semantics. For instance, the release and acquire memory orders allow programmers to derive release-acquire (RA) as well as its weak (WRA) and strong (SRA) variants [Lahav and Boker 2022]. The RA model is weaker than SC and provides a rigorous foundation in defining synchronization and locking primitives [Lahav et al. 2016]. The WRA and SRA are equivalent to variants of causal consistency [Lahav and Boker 2022], a well studied consistency model in the distributed systems literature. C11 also provides 'relaxed' memory access modes which constitutes the weaker memory model fragment Relaxed. Relaxed memory accesses can reorder freely and are the most performant compared to accesses with stronger memory orders. In our work, we focus on the recently proposed declarative RC20 memory model [Margalit and Lahav 2021] capturing a rich fragment of C11, consisting of release, acquire and relaxed memory accesses as well as memory fence operations. This memory model is a natural fragment of the C11 model, given that "*only a few (practical) algorithms that actually employ SC accesses and become wrong when release/acquire accesses are used instead*" [Margalit and Lahav 2021]. Further, focusing on the non-SC fragment allows us to reap the benefits of polynomial time consistency checking, which otherwise quickly becomes intractable [Gibbons and Korach 1997].

The intricacies of C11 and the abundance of its variants give rise to a plethora of consistency-checking instances. Some first results show that consistency checking for RA admits a polynomial bound [Abdulla et al. 2018; Lahav and Vafeiadis 2015], a stark difference to SC for which this problem is $\mathcal{NP}$-hard and is not even well-parameterizable [Gibbons and Korach 1997; Mathur et al. 2020]. These positive results have facilitated efficient model checking and testing techniques [Abdulla et al. 2018; Kokologiannakis et al. 2019; Luo and Demsky 2021]. However, beyond these recent developments, little is known about the complexity of consistency checking for C11-style memories, and, to our knowledge, the setting remains poorly understood. Our work fills this gap.

**Our contributions.** In this paper we study the reads-from consistency checking for the RA, SRA, WRA, Relaxed and RC20 memory models, with results that are optimal or nearly-optimal. In all cases, the input is a partial execution $\overline{\mathsf{X}} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf} \rangle$ with $n = |\mathsf{E}|$ events and $k$ threads, where po and rf are the program order and reads-from relation, respectively (see Section 2.1), and the task is to determine if there is a modification order mo such that the extension $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ is consistent with the memory model in consideration. Fig. 1 illustrates an example for RA-consistency.

Our first result concerns RC20. Consistency checking is known to be in polynomial time [Abdulla et al. 2018; Lahav and Vafeiadis 2015; Luo and Demsky 2021], though of degree 3 (i.e., $O(n^3)$). This cubic complexity has been identified as a challenge for efficient model checking (e.g., [Kokologiannakis et al. 2022, 2019]). Here we show that the full RC20 model admits an algorithm that is nearly linear-time; i.e., a bound that becomes linear when the number of threads is bounded.

Theorem 1.1. *Consistency checking for* RC20 *can be solved in* $O(n \cdot k)$ *time.*

A key step towards Theorem 1.1 is our notion of *minimal coherence*, which is a novel characterization that serves as a witness of consistency. Our consistency-checking algorithm proves consistency by constructing a minimally coherent (partial) modification order. Although similar witnesses have been used in the past (e.g., the writes-before order [Lahav and Vafeiadis 2015], saturated traces [Abdulla et al. 2018], or C11Tester's framework [Luo and Demsky 2021]), the simplicity of minimal coherence allows, for the first time to our knowledge, for a nearly linear-time algorithm.

Next we turn our attention to SRA. Perhaps surprisingly, although the model is conceptually close to RA, it turns out that checking consistency for SRA is intractable.

Theorem 1.2. *Consistency checking for* SRA *is* $\mathcal{NP}$*-complete, and* $\mathcal{W}[1]$*-hard in the parameter* $k$.

Here $\mathcal{W}[1]$ is a parameterized complexity class [Chen et al. 2004]. This result states that, not only is the problem $\mathcal{NP}$-complete, but it is also unlikely to be *fixed parameter tractable* in $k$, i.e., solvable in time $O(n^c \cdot f(k))$, where $c > 0$ and $f$ are independent of $n$. Nevertheless, our next result shows that this problem admits an upper bound that is polynomial when $k = O(1)$. Given the $\mathcal{W}[1]$-hardness, the next result is thus optimal, in the sense that $k$ has to appear in the exponent of $n$.

Theorem 1.3. *Consistency checking for* SRA *can be solved in* $O(k \cdot n^{k+1})$ *time.*

Taking a closer look into the model, we identify RMWs as the source of intractability. Indeed, the RMW-free fragment of SRA admits a nearly linear bound, much like RC20. This fragment is relevant, as it coincides with the causal convergence model [Bouajjani et al. 2017].

Theorem 1.4. *Consistency checking for the RMW-free fragment of* SRA *can be solved in* $O(n \cdot k)$ *time.*

Next, we show that the problem can be solved just as efficiently for WRA.

Theorem 1.5. *Consistency checking for* WRA *can be solved in* $O(n \cdot k)$ *time.*

Turning our attention to the Relaxed fragment of RC20, we show that the problem admits a truly linear bound (i.e., regardless of the number of threads).

Theorem 1.6. *Consistency checking for* Relaxed *can be solved in* $O(n)$ *time.*

Finally, observe that, in contrast to Theorem 1.6, the bounds in Theorem 1.1, Theorem 1.4 and Theorem 1.5 can become super-linear in the presence of many threads. It is thus tempting to search for a truly linear-time algorithm for any of RA, WRA and (RMW-free) SRA. Unfortunately, our final result shows that this is unlikely, in all models.

Theorem 1.7. *There is no consistency-checking algorithm for the RMW-free fragments of any of* RA*,* WRA*, and* SRA *that runs in time $O(n^{\omega/2-\epsilon})$, for any fixed $\epsilon > 0$. Moreover, there is no combinatorial algorithm for the problem that runs in time $O(n^{3/2-\epsilon})$, under the combinatorial BMM hypothesis.*

Here $\omega$ is the matrix multiplication exponent, with currently $\omega \simeq 2.37$. Theorem 1.7 states that a truly linear-time algorithm for any of these models would bring matrix multiplication in $n^{2+o(1)}$ time, a major breakthrough. Focusing on combinatorial algorithms (i.e., excluding algebraic fast-matrix multiplication, which appears natural in our setting), consistency checking for any of these models requires at least $n^{3/2}$ time unless (boolean) matrix multiplication (BMM) is improved below the classic cubic bound (which is considered unlikely, aka the BMM hypothesis [Williams 2019]).

Due to space restrictions, we relegate all proofs to to our technical report [Tunç et al. 2023a].

**Experiments.** We have implemented our algorithms inside the TruSt model checker and the C11Tester testing tool, and evaluated their performance on consistency checking for benchmarks utilizing instructions in the RA model. Our results report consistent and often significant speedups that reach 162× for TruSt and 104.2× for C11Tester.

Overall, our efficiency results enable practitioners to perform model checking and testing for RC20, RMW-free SRA, WRA, and Relaxed more efficiently, and apply these techniques to larger systems. On the other hand, our hardness result for SRA indicates that, akin to SC, performing consistency checking for SRA efficiently requires developing practically oriented heuristics that work well in the common cases. Finally, our super-linear lower bound for all models except Relaxed indicates that further improvements over our $O(n \cdot k)$ bounds will likely be highly non-trivial.

## 2 AXIOMATIC CONCURRENCY SEMANTICS

In this section we introduce the C/C++ concurrency semantics we consider in this work, along with the RC20 model and its variants [Lahav and Boker 2022; Margalit and Lahav 2021].

**Syntax.** C/C++ defines a shared memory concurrency model using different kinds of concurrency primitives. In addition to plain (or non-atomic) load and store accesses, C/C++ provides atomic accesses for load, store, atomic read-modify-write (RMW – such as atomic increment), and fence operations. We only consider atomic accesses here. An atomic access is parameterized by a memory mode, among relaxed (rlx), acquire (acq), release (rel), acquire-release (acq-rel), and sequential-consistency (sc). The memory order for a read, write, RMW, and fence access is one of {rlx, acq, sc}, {rlx, rel, sc}, {rlx, acq, rel, acq-rel, sc}, and {acq, rel, acq-rel, sc}, respectively. These accesses result in different types of events during execution. In this paper we consider the models which are based on non-SC primitives. Nevertheless, RC20 defines SC fences using the release-acquire primitives [Lahav and Boker 2022]. The memory modes are partially ordered on increasing strength of synchronization according to the lattice rlx $\sqsubset$ {acq, rel} $\sqsubset$ acq-rel. An access is *acquire* (*release*) if its order is acq (rel), or stronger.

### 2.1 Executions

The axiomatic concurrency models are defined with respect to the executions they allow. Hence, a program can be represented as a set of executions. In turn, an execution is defined by a set of events that are generated from shared memory accesses or fences, and relations between these events.

**Events.** An event is a tuple $\langle \mathsf{id}, \mathsf{tid}, \mathsf{lab} \rangle$ where id, tid, lab denote a unique identifier, thread identifier, and label of the event. The label $\mathsf{lab} = \langle \mathsf{op}, \mathsf{ord}, \mathsf{loc} \rangle$ is a tuple where op denotes the corresponding

$$\mathsf{fr} \triangleq (\mathsf{rf}^{-1};\mathsf{mo}) \setminus [\mathsf{id}]$$
$$\mathsf{sw} \triangleq [\mathsf{E}^{\sqsupseteq\textsc{rel}}];([\mathsf{F}];\mathsf{po})^?;\mathsf{rf}^+;(\mathsf{po};[\mathsf{F}])^?;[\mathsf{E}^{\sqsupseteq\textsc{acq}}]$$
$$\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+ \qquad \mathsf{hbloc} \triangleq \textstyle\bigcup_x \mathsf{hb}_x$$

| | |
|---|---|
| $\mathrm{irr}(\mathsf{mo};\mathsf{rf}^?;\mathsf{hb}^?)$ | (Wcoh) |
| $\mathrm{acy}(\mathsf{hb} \cup \mathsf{mo})$ | (strong-Wcoh) |
| $\mathrm{irr}(\mathsf{fr};\mathsf{rf}^?;\mathsf{hb})$ | (Rcoh) |
| $\mathrm{irr}(\mathsf{hbloc};[\mathsf{W} \cup \mathsf{RMW}];\mathsf{hb};\mathsf{rf}^{-1})$ | (weak-Rcoh) |
| $\mathrm{irr}(\mathsf{fr};\mathsf{mo})$ | (atomicity) |
| $\forall u_1,u_2 \in \mathsf{RMW}, \mathsf{rf}(u_1) \neq \mathsf{rf}(u_2)$ | (weak-atomicity) |
| $\mathrm{acy}(\mathsf{po} \cup \mathsf{rf})$ | (PO-RF) |

| | |
|---|---|
| $\mathrm{acy}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr})$ | (SC) |
| (PO-RF) ∧ (Wcoh) ∧ (Rcoh) ∧ (atomicity) | (RA)† |
| (PO-RF) ∧ (strong-Wcoh) ∧ (Rcoh) ∧ (atomicity) | (SRA) |
| (PO-RF) ∧ (weak-Rcoh) ∧ (weak-atomicity) | (WRA) |
| (PO-RF) ∧ (Wcoh) ∧ (Rcoh) ∧ (atomicity) | (RC20) |
| (PO-RF) ∧ (Wcoh) ∧ (Rcoh) ∧ (atomicity) | (Relaxed)† |

† These are fragments of RC20.

Fig. 2. Relations, Axioms, and Consistency Models on C11 Concurrency.

memory access or fence operation and ord denotes the memory mode. For memory accesses, loc denotes its memory location, while in the case of fences, we have loc $=\perp$. For the purpose of the reads-from consistency problem we consider in this paper, we omit the *values* read or written in memory access events. We write $\mathtt{w}(t,x)/\mathtt{r}(t,x)/\mathtt{rmw}(t,x)$ to denote a write/read/read-modify-write event in thread $t$ on location $x$, and simply write $e(x)$ to denote an event for which the thread is implied or not relevant. As a matter of convention, we omit mentioning the memory order throughout the paper, as it will either be clear from the context or not relevant.

The set of read, write, atomic update, and fence events are R, W, RMW and F respectively, and are generated from the executions of load, store, atomic load store, fence accesses respectively. As we only deal with executions (as opposed to program source), we use RMW to denote a successful read-modify-write operation. Failed read-modify-write operations simply result in read accesses. We refine the set of events in various ways. For instance, $\mathsf{E}^{\sqsupseteq\textsc{rel}}$ denotes the set of events with memory order that is at least as strong as $\textsc{rel}$. For a set of events E, we write E.*locs*, $\mathsf{E}_x$, and E.*tids* to denote the set of distinct locations accessed by events in E, the subset of events in E that access memory location $x$, and the different threads participating in E.

**Notation on relations.** Consider a binary relation $S$ over a set of events E. The reflexive, transitive, reflexive-transitive closures, and inverse relations of $S$ are denoted as $S^?$, $S^+$, $S^*$, $S^{-1}$ respectively. The relation $S$ is acyclic if $S^+$ is irreflexive. We write $\mathrm{irr}(S)$ and $\mathrm{acy}(S)$ to denote that relation $S$ irreflexive and acyclic respectively. Given two relations $S_1$ and $S_2$, we denote their composition by $S_1;S_2$. $[A]$ denotes the identity relation on a set $A$, i.e. $[A](x,y) \triangleq x = y \land x \in A$. For a given memory location $x$, we let $S_x = [\mathsf{E}_x];S;[\mathsf{E}_x]$ be the restriction of $S$ to all events of E on $x$.

**Candidate executions and relations.** An execution (also referred to as candidate execution [Batty et al. 2011] or execution graph [Lahav et al. 2017]) is a tuple $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}\rangle$ where X.E is a set of events and X.po, X.rf, X.mo are binary relations over X.E. In particular, the *program order* po is a partial order that enforces a total order on events of the same thread. The *reads-from relation* $\mathsf{rf} \subseteq (\mathsf{W} \cup \mathsf{RMW}) \times (\mathsf{R} \cup \mathsf{RMW})$ associates write/RMW events $e_1$ to read/RMW events $e_2$ reading from $e_1$. Naturally, we require that $e_1.\mathsf{loc} = e_2.\mathsf{loc}$, and that $\mathsf{rf}^{-1}$ is a function, i.e., every read/RMW event has a unique writer. The *modification order* $\mathsf{mo} \subseteq (\mathsf{W} \cup \mathsf{RMW}) \times (\mathsf{W} \cup \mathsf{RMW})$ is the union of modification orders $\mathsf{mo}_x$, where each $\mathsf{mo}_x$ is a total order over $(\mathsf{W}_x \cup \mathsf{RMW}_x)$.

We frequently also use some derived relations (see Fig. 2). The *from-read relation* $\mathsf{fr} \subseteq (\mathsf{R} \cup \mathsf{RMW}) \times (\mathsf{W} \cup \mathsf{RMW})$ relates every read or RMW event to all the write or RMW events that are mo-after its own writer. The *synchronizes-with relation* $\mathsf{sw} \subseteq \mathsf{E}^{\sqsupseteq\textsc{rel}} \times \mathsf{E}^{\sqsupseteq\textsc{acq}}$ relates release and acquire events, for instance, when an acquire read event reads from a release write. Fence instructions combined with relaxed accesses also participate in sw. The *happens-before relation* hb is the transitive closure of po and sw. We also project hb to individual locations, denoted as hbloc.

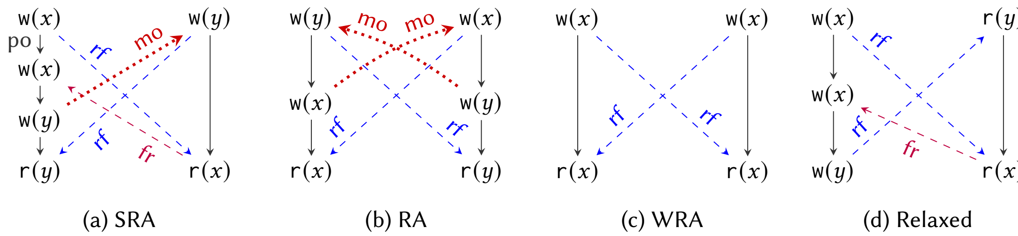



Fig. 3. Executions consistent in various memory models. mo edges that go along (po ∪ rf) are not shown.

## 2.2 Consistency Axioms

Consistency axioms characterize different aspects or constraints of a memory model. We broadly classify these constraints as coherence, atomicity, global ordering, and causality cycles. Different interpretations of these constraints give rise to different consistency models as shown in Fig. 2.

**Coherence.** In an execution, *coherence* enforces 'SC-per-location': the memory accesses per memory locations are totally ordered. *Write-coherence* enforces that mo agrees with hb. A stronger variant is *strong-write-coherence*, which requires that this condition holds transitively. *Read coherence* enforces that a read $\mathtt{r}(x)$ cannot read from a write $\mathtt{w}(x)$ if there is an *intermediate* write $\mathtt{w}'(x)$ that happens-before $\mathtt{r}(x)$, i.e. $\mathsf{hb}(\mathtt{w}'(x), \mathtt{r}(x))$ holds. In the vanilla read-coherence, the notion of 'intermediate' relates to mo, i.e., we have $(\mathtt{w}(x), \mathtt{w}'(x)) \in \mathsf{mo}$, while in *weak-read-coherence* [Lahav and Boker 2022], 'intermediate' relates to hb, i.e., we have $(\mathtt{w}(x), \mathtt{w}'(x)) \in \mathsf{hb}$.

**Atomicity.** The property ensures that (successful) RMW accesses indeed update the memory locations atomically. Following [Lahav and Boker 2022], we consider two variants. *Atomicity* ensures that no intermediate write event on the same location takes place between an RMW and its writer. *Weak-atomicity* simply prohibits two RMW events to have the same writer.

**Causality cycles.** A causality cycle arises in the presence of primitives that have weaker behaviors than release-acquire accesses. Such a cycle consists of po and rf orderings and may result in 'out-of-thin-air' behavior in certain programs. To avoid such 'out-of-thin-air' behavior, many consistency models explicitly disallow such cycles [Lahav and Boker 2022; Luo and Demsky 2021]. In the absence of RLX accesses, the PO-RF axiom coincides with the requirement for hb acyclicity.

## 2.3 Axiomatic Consistency Models and Consistency Checking

We can now present the memory models we consider in this work. See Fig. 2 for a summary.

**Sequential consistency.** Sequential consistency (SC) enforces a global order on all memory accesses. This is a stronger constraint than coherence, which orders same-location memory accesses. In addition, SC also guarantees atomicity.

**Release-Acquire and variants.** The release-acquire (RA) memory model is weaker than SC, and is arguably the most well-understood fragment of C11. At the same time, RA enables high-performance implementations of fundamental concurrency algorithms [Desnoyers et al. 2011; Lahav et al. 2016]. Broadly, under the release-acquire semantics (including other related variants), each read-from ordering establishes a synchronization. In this case hb reduces to $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$. Following [Lahav and Boker 2022], we consider three variants of release-acquire models: RA, strong RA (SRA), and weak RA (WRA). These models coincide with standard variants of causal consistency [Bouajjani et al. 2017; Burckhardt 2014; Lloyd et al. 2011], which is a ubiquitous consistency model relevant also in other domains such as distributed systems.

SRA enforces a stronger coherence guarantee (namely strong-write-coherence) on write accesses compared to RA. WRA does not place any restrictions on the mo ordering between same-location writes. Instead, the only orderings considered between same-location writes are through the [W]; hbloc; [W] relation. Thus, WRA provides weaker constraints for coherence and atomicity.

**RC20.** The recently introduced RC20 model [Margalit and Lahav 2021] defines a rich fragment of the C11 model consisting of acquire/release and relaxed accesses. Despite the absence of SC accesses, RC20 can express many practical synchronization algorithms, and can simulate SC fences.

**Relaxed.** Finally, the relaxed fragment of RC20 contains only RLX accesses, resulting in hb = po.

**Comparison between memory models.** The above models can be partially ordered according to the behaviors (executions) they allow as SC $\preccurlyeq$ SRA $\preccurlyeq$ RA $\preccurlyeq$ {WRA, {RC20 $\preccurlyeq$ Relaxed}}, with models towards the right allowing more behaviors. All models are weaker than SC. Relaxed is weaker than RA but incomparable with WRA. In particular, the lack of synchronization across rf in Relaxed makes hb weaker in Relaxed compared to WRA. On the other hand, WRA allows extra behaviors over Relaxed because it does not enforce write-coherence. Finally, RC20 can be viewed as a combination of RA and Relaxed, where fences may add synchronization between relaxed accesses. See Fig. 3 for an illustration.

**Extensions of the models with non-atomics.** For ease of presentation, we do not explicitly handle non-atomic accesses. The above memory models can be straightforwardly extended to include non-atomics with "catch-fire" semantics, similarly to previous works [Lahav and Boker 2022]. Intuitively, non-atomic accesses on any given location must always be hb-ordered, as otherwise this implies a data race, leading to undefined behavior [ISO/IEC 14882 2011; ISO/IEC 9899 2011].

**The reads-from consistency problem.** An execution $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ is *consistent* in a memory model $\mathcal{M}$, written $\mathsf{X} \models \mathcal{M}$, if it satisfies the axioms of the model. A *partial execution* $\overline{\mathsf{X}} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf} \rangle$ is an abstraction of executions without the modification order. We call $\overline{\mathsf{X}}$ *consistent* in $\mathcal{M}$, written similarly as $\overline{\mathsf{X}} \models \mathcal{M}$, if there exists an mo such that $\mathsf{X} \models \mathcal{M}$, where $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$. Thus the problem of *reads-from consistency checking* (or simply consistency checking, from now on) is to find an mo that turns $\overline{\mathsf{X}}$ consistent[*].

## 3 AUXILIARY FUNCTIONS, DATA STRUCTURES AND OBSERVATIONS

Our consistency checking algorithms rely on some common notation and computations. To avoid repetition, we present these here as auxiliary functions, while we refer to Fig. 4 for examples.

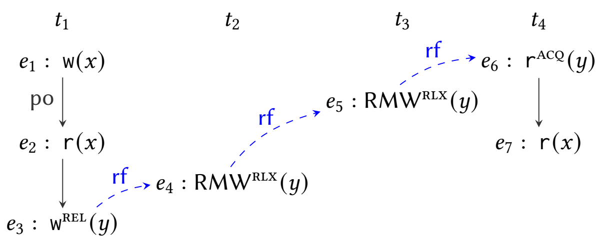
**Happens-before computation.** One common component in most of our algorithms is the computation of the hb-timestamp $\mathbb{HB}_e : \mathsf{E}.tids \to \mathbb{N}_{\geq 0}$ of each event $e$, declaratively defined as

$$\mathbb{HB}_e(t) = |\{f \mid f.\mathsf{tid} = t \land (f, e) \in \mathsf{hb}^?\}|$$

That is, $\mathbb{HB}_e(t)$ points to the last event of thread $t$ that is hb-ordered before (and including) $e$. The computation of all $\mathbb{HB}_e$ can be computed by a relatively straightforward algorithm (see e.g., [Luo and Demsky 2021]). We will thus take the following proposition for granted in this work.

Proposition 3.1. *The happens-before relation can be computed in $O(n \cdot k)$ time.*

[*] Except for WRA, the axioms of which do not involve mo.




$\mathbb{HB}_{e_7}(t_1) = 3,\ \mathbb{HB}_{e_7}(t_4) = 2$
$\mathbb{HB}_{e_7}(t_2) = \mathbb{HB}_{e_7}(t_3) = 0$
$\mathsf{lastWriteBefore}(t_1, x, 2) = e_1$
$\mathsf{lastWriteBefore}(t_2, x, 1) = \bot$
$\mathsf{lastWriteBefore}(t_2, y, 1) = e_4$
$\mathsf{lastReadBefore}(t_1, x, 2) = e_2$
$\mathbb{TC}[e_4] = \mathbb{TC}[e_5] = e_3$
$\mathbb{PC}[e_3] = 0,\ \mathbb{PC}[e_4] = 1,\ \mathbb{PC}[e_5] = 2$

Fig. 4. Example of a partial execution (left) and its auxiliary functions (right).

**Last write and last read.** Given a thread $t$, location $x$, and index $c \in \mathbb{N}_{\geq 0}$, we define

$$\mathsf{lastWriteBefore}(t, x, c) = \begin{cases} e & \text{if } e \text{ is the last event such that, } e \in \mathsf{W}_x \cup \mathsf{RMW}_x, \\ & e.\mathsf{tid} = t \text{ and } |\{g \mid (g, e) \in \mathsf{po}^?\}| \leq c \\ \bot & \text{if no such event exists} \end{cases}$$

$$\mathsf{lastReadBefore}(t, x, c) = \begin{cases} e & \text{if } e \text{ is the last event such that, } e \in \mathsf{R}_x \cup \mathsf{RMW}_x, \\ & e.\mathsf{tid} = t \text{ and } |\{g \mid (g, e) \in \mathsf{po}^?\}| \leq c \\ \bot & \text{if no such event exists} \end{cases}$$

In words, $\mathsf{lastWriteBefore}(t, x, c)$ returns the latest po-predecessor $\mathtt{w}(t, x)$ or $\mathtt{rmw}(t, x)$ of the $c$-th event of thread $t$ (similarly for $\mathsf{lastReadBefore}(t, x, c)$). When our consistency algorithms process a read/RMW event $e(u, x)$, they query for $\mathsf{lastWriteBefore}(t, x, c)$ and $\mathsf{lastReadBefore}(t, x, c)$ on each thread $t$, where $c = \mathbb{HB}_e(t)$. We call $u$ the *observer thread*. Our efficient handling of such queries is based on the insight that, along subsequent queries from the same observer thread, $c$ is monotonically increasing ($\mathbb{HB}$ timestamps are monotonic along po-ordered events). We develop a simple data structure for handling such queries as follows. For each thread $t$, memory location $x$, and observer thread $u$, we maintain lists $\mathbb{WList}^u_{t,x}$ and $\mathbb{RList}^u_{t,x}$, each containing the sequence of write/RMW events and read/RMW events performed by $t$ on $x$, together with their thread-local indices. The parameterization by $u$ ensures that $u$ observes its own local version of this list. Answering a query $\mathsf{lastWriteBefore}(t, x, c)$ consists of iterating over $\mathbb{WList}^u_{t,x}$ until the correct event is identified. Subsequent queries continue the iteration from the last returned position. The total cost of traversing all these lists is $O(n \cdot k)$ (each event appears in $k$ lists, one per observer thread). In pseudocode descriptions, we will call $\mathbb{WList}^u_{t,x}$`.get(c)` (resp., $\mathbb{RList}^u_{t,x}$`.get(c)`) to access the event $e = \mathsf{lastWriteBefore}(t, x, c)$ (resp., $e = \mathsf{lastReadBefore}(t, x, c)$). This implementation of lastWriteBefore and lastReadBefore is novel compared to prior works (e.g., [Luo and Demsky 2021]), and crucial for obtaining the linear-time bounds developed in our work.

**Top of, and position in rf-chain.** All memory models we consider satisfy weak-atomicity (atomicity implies weak atomicity), i.e., no two RMW events can have the same writer. This implies that all write and RMW events are arranged in disjoint rf*-chains*, where a chain is a maximal sequence of events $e_0, e_1, \ldots, e_\ell$ ($\ell \geq 0$), such that (i) $e_0 \in \mathsf{W}$ and $e_1, \ldots, e_\ell \in \mathsf{RMW}$, and (ii) $\mathsf{rf}^{-1}(e_i) = e_{i-1}$ for each $i \geq 1$. In words, we have a chain of events connected by rf, starting with the top write event $e_0$, and (optionally) continuing with a maximal sequence of RMW events that read from this chain. Given an event $e \in (\mathsf{W} \cup \mathsf{RMW})$, we often refer to the rf-chain that contains $e$. Specifically, the top of the chain $\mathbb{TC}[e]$ is the unique event $f$ such that $(f, e) \in \mathsf{rf}^*$ and $f.\mathsf{op} = \mathtt{w}$. The position $\mathbb{PC}[e]$ of $e$ in its rf-chain is $|\{f \mid (f, e) \in \mathsf{rf}^+\}|$. Both $\mathbb{TC}$ and $\mathbb{PC}$ can be computed in $O(n)$ time for all events.

**Algorithm 1:** Checking Consistency for WRA.

**Input:** A partial execution $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$

1 **if** $(\mathsf{po} \cup \mathsf{rf})$ *is cyclic or* rf *violates weak-atomicity* **then declare** 'Inconsistent'
2 **let** $\mathbb{HB}$ be an E-indexed array storing the hb-timestamps of events
3 **let** $\{\mathbb{WList}^{u}_{t,x}\}_{t,x,u}$ be data structures implementing lastWriteBefore$(\cdot,\cdot,\cdot)$
4 **foreach** $e \in \mathsf{E}$ *in* po*-order* **do**
5   **case** $e = \mathtt{r}(t,x)$ *or* $e = \mathtt{rmw}(t,x)$ **do**
6     **let** $w = \mathsf{rf}^{-1}(e)$, $t' = w.\mathrm{tid}$ and $c = \mathbb{HB}[w][t']$
7     **foreach** $u \in \mathsf{E}.tids$ **do**
8       **let** $c_u =$ **if** $(e.\mathrm{op} = \mathtt{rmw} \wedge u = t)$ **then** $\mathbb{HB}[e][u] - 1$ **else** $\mathbb{HB}[e][u]$
9       **let** $w_u = \mathbb{WList}^{u}_{t,x} \bullet \mathtt{get}(c_u)$
10       **if** $(\mathbb{HB}[w_u][t'] \geq c) \wedge ((u = t') \Rightarrow \mathbb{HB}[w_u][t'] > c)$ **then declare** 'Inconsistent'
11 **declare** 'Consistent'

**Conflicting triplets.** A *conflicting triplet* (or just *triplet*, for short) is a triplet of distinct events $(e_1, e_2, e_3)$ such that (i) all events access the same location $x$, (ii) $e_1, e_3 \in (\mathsf{W}_x \cup \mathsf{RMW}_x)$ and $e_2 \in (\mathsf{R}_x \cup \mathsf{RMW}_x)$, and (iii) $\mathsf{rf}^{-1}(e_2) = e_1$.

Finally, we state a simple lemma that is instrumental throughout the paper. This lemma identifies certain mo orderings implied by the basic axioms of read-coherence, write-coherence and atomicity, and thus applies to all models except WRA. Fig. 5 provides an illustration.

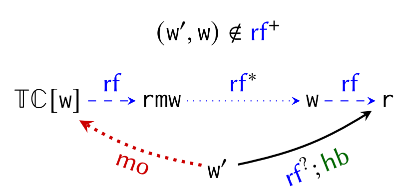



Fig. 5. An mo ordering implied by read-coherence, write-coherence and atomicity.

Lemma 3.2. *Let* $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ *be an execution that satisfies read-coherence, write-coherence and atomicity. Let* $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$ *be a triplet. If* $(\mathtt{w}', \mathtt{r}) \in \mathsf{rf}^{?}; \mathsf{hb}$ *and* $(\mathtt{w}', \mathtt{w}) \notin \mathsf{rf}^{+}$*, then* $(\mathtt{w}', \mathbb{TC}[\mathtt{w}]) \in \mathsf{mo}$.

## 4 CONSISTENCY CHECKING

This section presents the main results of the paper, as outlined in Section 1. We start with an algorithm for checking consistency under WRA in Section 4.1. For SRA, we show in Section 4.2 that the problem is $\mathcal{NP}$-complete in general, but has a polynomial time algorithm for the RMW-free programs as shown in Section 4.3. Section 4.4 and Section 4.5 show that consistency checking is polynomial time for RC20 and linear-time for Relaxed along with the respective algorithms. Finally, we study the lower bound of consistency checking for RMW-free RA, WRA, and SRA in Section 4.6.

### 4.1 Consistency Checking for WRA

We start with the WRA model, which is conceptually simpler as there is no mo involved in the consistency axioms. Algorithm 1 checks for consistency in $O(n \cdot k)$, towards Theorem 1.5.

Given a partial execution $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$, the algorithm first verifies that there are no $(\mathsf{po} \cup \mathsf{rf})$-cycles and every write/RMW event is read by at most one RMW event (Line 1). Afterwards, the algorithm streams the events of E in an order consistent with po and verifies weak-read-coherence, i.e., there is no triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$ such that $\{(\mathtt{w}, \mathtt{w}'), (\mathtt{w}', \mathtt{r})\} \subseteq \mathsf{hb}$. In order to check this in linear time, the algorithm first computes the array of hb-timestamps (Line 2) and the last write events lastWriteBefore$(t, x, c)$ for each thread $t$, location $x$ and index $c$ (Line 3), as defined in Section 3.

In order to check that weak-read-coherence is not violated, at a read/RMW event $e$ with $e$.tid $= t$ and $e$.loc $= x$, Algorithm 1 checks if there is an event $e' \in \mathsf{W}_x \cup \mathsf{RMW}_x$ such that $e'$ is hb-sandwiched between $\mathsf{rf}^{-1}(e)$ and $e$. Since $\mathsf{po} \subseteq \mathsf{hb}$, it suffices to check if for any thread $u$, the event lastWriteBefore$(u, x, \mathbb{HB}_e[u])$ can play the role of $e'$ above (Line 10).

The total running time on an input partial execution $\overline{\mathsf{X}}$ with $n$ events and $k$ threads can be computed as follows. The initialization of $\mathbb{HB}$ and the lists $\{\mathbb{WList}^u_{t,x}\}_{t,x,u}$, and the total cost of all calls to $\mathbb{WList}^u_{t,x}$ • `get(`$c_u$`)` takes $O(n \cdot k)$ time (Section 3). Afterwards, the algorithm spends $O(k)$ time at each event. This gives a total running time of $O(n \cdot k)$. We thus have the following theorem.

Theorem 1.5. *Consistency checking for* WRA *can be solved in* $O(n \cdot k)$ *time.*

### 4.2 Consistency Checking for SRA

We now turn our attention to SRA, and prove the bounds of Theorem 1.2 and Theorem 1.3.

**The hardness of consistency checking for** SRA**.** First, note that consistency checking is a problem in $\mathcal{NP}$. Indeed, given a partial execution $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$, one can guess a candidate mo and verify that $\mathsf{X} \models \mathsf{SRA}$, where $\mathsf{X} = (\mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$ is a complete execution, by checking against the axioms of SRA. Each axiom can be verified in polynomial time, giving us membership in $\mathcal{NP}$. Now we turn our attention to $\mathcal{W}[1]$-hardness (which will also imply $\mathcal{NP}$-hardness). Our reduction is from the consistency problem for SC, which is known to be $\mathcal{NP}$-hard [Gibbons and Korach 1997] and more recently shown to be $\mathcal{W}[1]$-hard [Mathur et al. 2020]. We obtain hardness in two steps.

First, we observe that the consistency problem for SC is $\mathcal{W}[1]$-hard even over instances in which every write event is observed at most once. This can be obtained from the proof of $\mathcal{W}[1]$-hardness in [Mathur et al. 2020]. Towards our $\mathcal{W}[1]$-hardness proof for SRA, we can substitute in such instances every read access by an RMW access without affecting the SC consistency of the execution. Intuitively, as any write observed by a read does not have any other readers, the write of the substituting RMW has no effect. Formally, we have the following lemma.

Lemma 4.1. *Consistency checking for* SC *with only write and RMW events is* $\mathcal{W}[1]$*-hard in the parameter* $k$*.*

Given Lemma 4.1, we can now prove Theorem 1.2. The key observation is that the strong-write-coherence of SRA implies a total order on all write/RMW events. Thus, over instances where every event is either a write or an RMW, strong-write-coherence yields a total order on all events, which, in turn, implies an SC-consistent execution. We arrive at the following theorem.

Theorem 1.2. *Consistency checking for* SRA *is* $\mathcal{NP}$*-complete, and* $\mathcal{W}[1]$*-hard in the parameter* $k$*.*

**A parameterized upper bound.** We now turn our attention to Theorem 1.3, i.e., we solve consistency checking for SRA in time $O(k \cdot n^{k+1})$. Recall that our goal is to construct an mo that witnesses the consistency of $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$. One natural approach is to enumerate all possible mo's and check whether any of them leads to a consistent X. However, this leads to an exponential algorithm regardless of the number of threads (there are exponentially many possible mo's even with two threads) which is beyond the bound of Theorem 1.3. We instead follow a different approach.

*Algorithm.* Given the poset $(\mathsf{E}, \mathsf{hb})$, a set $Y \subseteq \mathsf{E}$ is said to be *downward-closed* if for all $e_1 \in Y, e_2 \in \mathsf{E}$, if $(e_2, e_1) \in \mathsf{hb}$ then $e_2 \in Y$. We define a (directed) *downward graph* $\mathcal{G}_{\overline{\mathsf{X}}}$ induced by $(\mathsf{E}, \mathsf{hb})$, and show that the question of $\overline{\mathsf{X}} \models \mathsf{SRA}$ reduces to checking reachability in $\mathcal{G}_{\overline{\mathsf{X}}}$. The node set of $\mathcal{G}_{\overline{\mathsf{X}}}$ consists of all downward closed subsets $S$ of $\mathsf{E}$, with $\varnothing$ being the *root node* and $\mathsf{E}$ being the *terminal* *node*. Given a node $S$ in $\mathcal{G}_{\overline{X}}$, we insert edges $S \longrightarrow S'$ where $S'$ is obtained by extending $S$ with an event which is *executable* in $S$. An event $e$ executable in $S$ if the following conditions hold.

(1) All events $e'$ such that $(e', e) \in \mathsf{hb}$ are in $S$.
(2) If $e \in (\mathsf{W} \cup \mathsf{RMW})$ is a write/RMW event, then it must also be *enabled*. We say that a write/RMW event $\mathtt{w}$ is enabled if the following hold.
  (a) For every triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$, if $(\mathtt{w}', \mathtt{r}) \in \mathsf{hb}$, then $\mathtt{w}' \in S$. Intuitively, executing $\mathtt{w}$ while $\mathtt{w}' \notin S$ represents a guess that $(\mathtt{w}, \mathtt{w}') \in \mathsf{mo}$, which would violate read-coherence as $(\mathtt{w}', \mathtt{r}) \in \mathsf{hb}$.
  (b) For every RMW event $\mathtt{rmw}$ and triplet $(\mathtt{w}', \mathtt{rmw}, \mathtt{w})$, if $\mathtt{rmw} \notin S$ then $\mathtt{w}' \notin S$. Intuitively, executing $\mathtt{w}$ while $\mathtt{rmw} \notin S$ but $\mathtt{w}' = \mathsf{rf}^{-1}(\mathtt{rmw}) \in S$ represents a guess that $(\mathtt{w}', \mathtt{w}) \in \mathsf{mo}$ and $(\mathtt{w}, \mathtt{rmw}) \in \mathsf{mo}$, which would violate atomicity as it would imply $(\mathtt{rmw}, \mathtt{w}) \in \mathsf{fr}$ and thus $(\mathtt{rmw}, \mathtt{rmw}) \in \mathsf{fr}; \mathsf{mo}$.

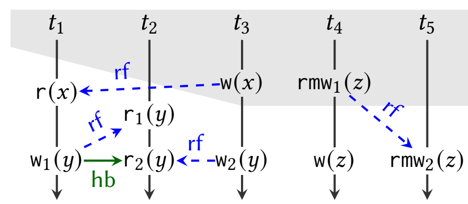



Fig. 6. Enabledness and executability of a set $S$, marked in gray. Only $\mathtt{r}(x)$ and $\mathtt{rmw}_2(z)$ are executable events. $\mathtt{w}_1(y)$ is not executable as $\mathtt{r}(x) \notin S$. $\mathtt{r}_1(y)$ is not executable as $\mathtt{w}_1(y) \notin S$. $\mathtt{w}_2(y)$ is not enabled as $(\mathtt{w}_1(y), \mathtt{r}_2(y)) \in \mathsf{hb}$ but $\mathtt{w}_1(y) \notin S$. $\mathtt{w}(z)$ is not enabled as $\mathtt{rmw}_2(z) \notin S$.

Fig. 6 illustrates the above notions. Conceptually, every path from the root $\varnothing$ to a node $S$ in $\mathcal{G}_{\overline{X}}$ represents an $\mathsf{mo}$ on the write/RMW events of $S$. Although there can be exponentially many such paths, the node $S$ "forgets" their corresponding $\mathsf{mo}$'s. Instead, $S$ represents a partial $\overline{\mathsf{mo}}$, which orders every write/RMW event on a location $x$ of $S$ before every write/RMW event on $x$ outside $S$. The algorithm terminates and returns that $\overline{\mathsf{X}} \models \mathsf{SRA}$ iff the node $\mathsf{E}$ is reachable from the root node $\varnothing$ in $\mathcal{G}_{\overline{X}}$. Observe that $\mathcal{G}_{\overline{X}}$ contains $O(n^k)$ nodes, while each node has $\leq k$ successors. Deciding whether a node has a transition to another node can be easily done in $O(n)$ time. Hence, the total time of the algorithm is $O(k \cdot n^{k+1})$. We thus arrive at the following theorem.

Theorem 1.3. *Consistency checking for* SRA *can be solved in* $O(k \cdot n^{k+1})$ *time.*

### 4.3 Consistency Checking for the RMW-Free Fragment of SRA

On close inspection, RMW events played a central role in the $\mathcal{NP}$-hardness of Theorem 1.2. A natural question thus arises: does the hardness persist in the absence of RMWs? Here we show that the RMW-free fragment of SRA can be handled efficiently (Theorem 1.4). Observe that Lemma 3.2 applies to SRA, as strong-write-coherence implies write-coherence. However, the lemma admits a simplification under RMW-free SRA. In particular, as $\mathsf{X}$ does not contain RLX accesses, $(\mathtt{w}', \mathtt{r}) \in \mathsf{rf}^?; \mathsf{hb}$ reduces to $(\mathtt{w}', \mathtt{r}) \in \mathsf{hb}$. Moreover, as $\mathsf{X}$ is RMW-free, we have $\mathbb{TC}[\mathtt{w}] = \mathtt{w}$, and hence $(\mathtt{w}', \mathtt{w}) \in \mathsf{mo}$. Thus, Lemma 3.2 reduces to the following corollary.

Corollary 4.2. *Consider any execution* $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ *that satisfies read-coherence and strong-write-coherence. Consider any triplet* $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$. *If* $(\mathtt{w}', \mathtt{r}) \in \mathsf{hb}$ *then* $(\mathtt{w}', \mathtt{w}) \in \mathsf{mo}$.

**Minimal coherence under** SRA. Corollary 4.2 identifies necessary orderings in any modification order that witnesses the consistency of $\overline{\mathsf{X}}$. Towards an algorithm, we must also determine if there are non-trivial conditions *sufficient* to conclude consistency. We answer this in the positive, by capturing these conditions in the notion of *minimal coherence*. Consider a partial modification order $\overline{\mathsf{mo}} = \bigcup_x \overline{\mathsf{mo}}_x$, where each $\overline{\mathsf{mo}}_x$ is a partial order. We call $\overline{\mathsf{mo}}$ *minimally coherent for* $\overline{\mathsf{X}}$ *under* SRA if the following conditions hold.

(1) For every triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$ with $(\mathtt{w}', \mathtt{r}) \in \mathsf{hb}$, we have $(\mathtt{w}', \mathtt{w}) \in (\mathsf{hb} \cup \overline{\mathsf{mo}})^+$.
(2) $(\mathsf{hb} \cup \overline{\mathsf{mo}})$ is acyclic.

**Algorithm 2:** Checking consistency for the RMW-free fragment of SRA.

**Input:** Events $\mathsf{E}$, program order $\mathsf{po}$ and reads-from relation $\mathsf{rf}$

**if** $(\mathsf{po} \cup \mathsf{rf})$ *is cyclic* **then declare** 'Inconsistent'
**let** $\mathbb{HB}$ be an $\mathsf{E}$-indexed array storing the $\mathsf{hb}$-timestamps of events
**let** $\{\mathbb{WList}^{u}_{t,x}\}_{t,x,u}$ be data structures implementing $\mathrm{lastWriteBefore}(\cdot,\cdot,\cdot)$
**foreach** $x \in \mathsf{E}.locs$ **do** $\overline{\mathsf{mo}}_x \leftarrow \emptyset$
**foreach** $e \in \mathsf{E}$ *in* $\mathsf{po}$*-order* **do**
  **case** $e = \mathsf{r}(t, x)$ **do**
    **let** $\mathsf{w}_{\mathsf{rf}} = \mathsf{rf}^{-1}(e)$
    **foreach** $u \in \mathsf{E}.tids$ **do**
      **let** $\mathsf{w}_u = \mathbb{WList}^{t}_{u,x} \cdot \mathtt{get}(\mathbb{HB}[e][u])$
      **if** $\mathsf{w}_u \neq \mathsf{w}_{\mathsf{rf}}$ **then** $\overline{\mathsf{mo}}_x \leftarrow \overline{\mathsf{mo}}_x \cup \{(\mathsf{w}_u, \mathsf{w}_{\mathsf{rf}})\}$
**if** $(\mathsf{hb} \cup \bigcup_{x \in \mathsf{E}.locs} \overline{\mathsf{mo}}_x)$ *is cyclic* **then declare** 'Inconsistent'
**else declare** 'Consistent'

Fig. 7 illustrates the notion of minimal coherence under SRA. Observe that any $\mathsf{mo}$ witnessing the consistency of $\overline{\mathsf{X}}$ satisfies these conditions. In the following, we show that minimally coherent modification orders are also sufficient for witnessing consistency. We note that for RMW-free executions, minimal coherence coincides with the previous notions of coherence that witnesses consistency under RA [Abdulla et al. 2018; Lahav and Vafeiadis 2015; Luo and Demsky 2021]. However, these notions also handle RMWs, and are not directly applicable in SRA, as the problem of consistency checking is $\mathcal{NP}$-hard for SRA with RMWs (Theorem 1.2).

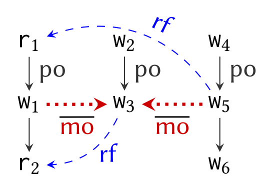



Fig. 7. A minimally coherent $\overline{\mathsf{mo}}$ under SRA. All events access the same location.

Lemma 4.3. *Consider any RMW-free, partial execution* $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$. *If there exists partial modification order* $\overline{\mathsf{mo}}$ *that is minimally coherent for* $\overline{\mathsf{X}}$ *under* SRA, *then* $\overline{\mathsf{X}} \models \mathsf{SRA}$.

**Algorithm.** Corollary 4.2 and Lemma 4.3 suggest a polynomial-time algorithm for deciding the SRA consistency of an RMW-free, partial execution $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$. Similarly to WRA, we first verify that $(\mathsf{po} \cup \mathsf{rf})$ is acyclic. Then, we construct a partial modification order $\overline{\mathsf{mo}}$ by identifying all conflicting triplets $(\mathsf{w}, \mathsf{r}, \mathsf{w}')$ such that $(\mathsf{w}', \mathsf{r}) \in \mathsf{hb}$, and inserting an ordering $(\mathsf{w}', \mathsf{w}) \in \overline{\mathsf{mo}}$. Finally, we report that $\overline{\mathsf{X}} \models \mathsf{SRA}$ iff $(\mathsf{hb} \cup \overline{\mathsf{mo}})$ is acyclic.

Although this process runs in polynomial time, it is still far from the nearly linear bound we aim for (Theorem 1.4). The key extra step towards this bound comes from a closer look at minimal coherence: based on Item (1), it suffices to only consider conflicting triplets $(\mathsf{w}, \mathsf{r}, \mathsf{w}')$ in which $\mathsf{w}'$ is $\mathsf{po}$-maximal among all write events $\mathsf{w}''$ forming a conflicting triplet $(\mathsf{w}, \mathsf{r}, \mathsf{w}'')$ such that $(\mathsf{w}'', \mathsf{r}) \in \mathsf{hb}$. For each thread $t$, we thus only need to identify the $\mathsf{po}$-maximal write $\mathsf{w}'$ in the scheme outlined above. This concept is illustrated in Fig. 7. Consider the triplets $(\mathsf{w}_3, \mathsf{r}_2, \mathsf{w}_4)$, $(\mathsf{w}_3, \mathsf{r}_2, \mathsf{w}_5)$, and $(\mathsf{w}_3, \mathsf{r}_2, \mathsf{w}_6)$. Only the first two satisfy the above definition (since $(\mathsf{w}_4, \mathsf{r}_2), (\mathsf{w}_5, \mathsf{r}_2) \in \mathsf{hb}$ but $(\mathsf{w}_6, \mathsf{r}_2) \notin \mathsf{hb}$). In this case, only identifying the event $\mathsf{w}_5$ is sufficient as it is the $\mathsf{po}$-maximal write among $\mathsf{w}_4$ and $\mathsf{w}_5$.

This insight is precisely formulated in Algorithm 2. The algorithm uses the auxiliary functions from Section 3 to compute the $\mathbb{HB}$-timestamp of each event. Also recall that, for threads $t$ and $u$ and location $x$, $\mathbb{WList}^{u}_{t,x}$ denotes (thread $u$'s copy of) the $\mathsf{po}$-ordered list of write accesses performed by $t$ on location $x$. It then processes events in $\overline{\mathsf{X}}$ in an order consistent with $\mathsf{po}$ and builds a partial modification order $\overline{\mathsf{mo}}$. When processing a read event $\mathsf{r}$, the algorithm identifies for every thread $u$, the $\mathsf{po}$-maximal write $\mathsf{w}'$ of $u$ that forms a conflicting triplet $(\mathsf{w}, \mathsf{r}, \mathsf{w}')$ with $(\mathsf{w}', \mathsf{r}) \in \mathsf{hb}$ (Line 9), and inserts $(\mathsf{w}', \mathsf{w}) \in \overline{\mathsf{mo}}$ (Line 10). Finally, it checks whether $\overline{\mathsf{mo}}$ violates strong-write-coherence.

**Correctness and complexity.** The completeness follows directly from Corollary 4.2: every ordering inserted in $\overline{\mathsf{mo}}_x$ is present in any modification order $\mathsf{mo}$ that witnesses the consistency of $\overline{\mathsf{X}}$, while the acyclicity check in Line 11 is necessary for strong-write-coherence. Hence, if the algorithm returns "Inconsistent", we have $\overline{\mathsf{X}} \not\models \mathsf{SRA}$. The soundness comes from the fact that $\overline{\mathsf{mo}}$ satisfies Item (1) of minimal coherence at the end of the loop of Line 5, while if the acyclicity check in Line 11 passes, $\overline{\mathsf{mo}}$ also satisfies Item (2) of minimal coherence. Thus, by Lemma 4.3, $\overline{\mathsf{X}} \models \mathsf{SRA}$.

The time spent in computing $\mathbb{HB}$ and initializing and accessing the lists $\mathbb{WList}^{u}_{t,x}$ is $O(n \cdot k)$ (Section 3). The number of orderings added in $\overline{\mathsf{mo}}$ is $O(n \cdot k)$, taking $O(n \cdot k)$ total time. Finally, the check in Line 11 is $O(n \cdot k)$ time as it corresponds to detecting a cycle on a graph with $|\mathsf{E}| = n$ nodes and $\leq n \cdot (k+1)$ edges. This gives a total running time of $O(n \cdot k)$. We thus arrive at Theorem 1.4.

Theorem 1.4. *Consistency checking for the RMW-free fragment of* SRA *can be solved in* $O(n \cdot k)$ *time.*

### 4.4 Consistency Checking for RC20

We now turn our attention to the full RC20 memory model, which comprises a mixture of REL, ACQ and RLX memory accesses. Similarly to the RMW-free SRA, we obtain a nearly linear bound (Theorem 1.1). Note, however, that here we also allow RMW events. As RC20 satisfies read-coherence, write-coherence and atomicity, Lemma 3.2 applies also in this setting. However, our earlier notion of minimal coherence under SRA is no longer applicable as is — Lemma 4.3 does not hold for RC20. Fortunately, we show this model enjoys a similar notion of coherence minimality.

**Minimal coherence under** RC20. Consider a partial modification order $\overline{\mathsf{mo}} = \bigcup_x \overline{\mathsf{mo}}_x$. We call $\overline{\mathsf{mo}}$ *minimally coherent for* $\overline{\mathsf{X}}$ *under* RC20 if the following conditions hold.

(1) For every triplet $(\mathsf{w}, \mathsf{r}, \mathsf{w}')$ accessing location $x$, if $(\mathsf{w}', \mathsf{r}) \in \mathsf{rf}^?; \mathsf{hb}$ and $(\mathsf{w}', \mathsf{w}) \notin \mathsf{rf}^+$, then $(\mathsf{w}', \mathbb{TC}[\mathsf{w}]) \in (\mathsf{rf}_x \cup \mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)^+$.
(2) For every two write/RMW events $\mathsf{w}_1, \mathsf{w}_2$ accessing location $x$, if $(\mathsf{w}_1, \mathsf{w}_2) \notin \mathsf{rf}^+$ and $(\mathsf{w}_1, \mathsf{w}_2) \in \overline{\mathsf{mo}}_x$, then $(\mathsf{w}_1, \mathbb{TC}[\mathsf{w}_2]) \in \overline{\mathsf{mo}}_x$.
(3) $(\mathsf{rf}_x \cup \mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)$ is acyclic, for each $x \in \mathsf{E}.locs$.

Fig. 8 illustrates the above definition. Observe that any $\mathsf{mo}$ witnessing the consistency of $\overline{\mathsf{X}}$ satisfies minimal coherence. As before, minimal coherence is also a sufficient witness of consistency.

Lemma 4.4. *Consider any partial execution* $\overline{\mathsf{X}} = (\mathsf{E}, \mathsf{po}, \mathsf{rf})$. *If there exists partial modification order* $\overline{\mathsf{mo}}$ *that is minimally coherent for* $\overline{\mathsf{X}}$ *under* RC20, *then* $\overline{\mathsf{X}} \models$ RC20.

Our algorithm for consistency checking in RC20 relies on Lemma 4.4 to construct a minimally-coherent partial modification order that witnesses the consistency of $\overline{\mathsf{X}}$. In particular, the algorithm employs the simple inference rule of $\overline{\mathsf{mo}}$ edges illustrated earlier in Fig. 5, and is a direct application of Item (1) of minimal coherence. At a glance, it might come as a surprise that such a simple rule suffices to deduce consistency. Indeed, analogous relations have been used in the past as consistency witnesses (e.g., the writes-before order [Lahav and Vafeiadis 2015], saturated traces [Abdulla et al. 2018], or C11Tester's framework [Luo and Demsky 2021]).

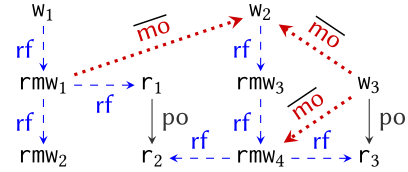



Fig. 8. A minimally coherent $\overline{\mathsf{mo}}$ under RC20. All events access the same location.

**Algorithm 3:** Checking consistency for RC20.

**Input:** Events $\mathsf{E}$, program order $\mathsf{po}$ and reads-from relation $\mathsf{rf}$

1 **if** $(\mathsf{po} \cup \mathsf{rf})$ *is cyclic or* $\mathsf{rf}$ *violates weak-atomicity* **then** **declare** 'Inconsistent'

2 **let** $\mathbb{HB}$ be an $\mathsf{E}$-indexed array storing the $\mathsf{hb}$-timestamps of events

3 **let** $\{\mathbb{WList}^u_{t,x}\}_{t,x,u}$ and $\{\mathbb{RList}^u_{t,x}\}_{t,x,u}$ be data structures implementing lastWriteBefore() and lastReadBefore()

4 **let** $\mathbb{TC}$ and $\mathbb{PC}$ be $\mathsf{E}$-indexed arrays denoting the top and position of events in their $\mathsf{rf}$-chains

5 **foreach** $x \in \mathsf{E}.locs$ **do** $\overline{\mathsf{mo}}_x \leftarrow \emptyset$;

6 **foreach** $e \in \mathsf{E}$ *in* $\mathsf{po}$*-order* **do**

7     **case** $e = \mathtt{r}(t, x)$ *or* $e = \mathtt{rmw}(t, x)$ **do**

8         **let** $\mathtt{w}_{\mathsf{rf}} = \mathsf{rf}^{-1}(e)$

9         **foreach** $u \in \mathsf{E}.tids$ **do**

10             **let** $c_u =$ **if** $(e.\mathrm{op} = \mathtt{rmw} \wedge u = t)$ **then** $\mathbb{HB}[e][u] - 1$ **else** $\mathbb{HB}[e][u]$

11             **let** $\mathtt{w}^{\mathtt{w}}_u = \mathbb{WList}^t_{u,x} \cdot \mathtt{get}(c_u)$ and **let** $\mathtt{w}^{\mathtt{r}}_u = \mathsf{rf}^{-1}(\mathbb{RList}^t_{u,x} \cdot \mathtt{get}(c_u))$

12             **for** $\mathtt{w}_u \in \{\mathtt{w}^{\mathtt{w}}_u, \mathtt{w}^{\mathtt{r}}_u\}$ **do**

13                 **if** $(\mathbb{TC}[\mathtt{w}_{\mathsf{rf}}] \neq \mathbb{TC}[\mathtt{w}_u])$ *or* $(\mathbb{PC}[\mathtt{w}_{\mathsf{rf}}] < \mathbb{PC}[\mathtt{w}_u])$ **then**

14                     $\overline{\mathsf{mo}}_x \leftarrow \overline{\mathsf{mo}}_x \cup \{(\mathtt{w}_u, \mathbb{TC}[\mathtt{w}_{\mathsf{rf}}])\}$

15 **foreach** $x \in \mathsf{E}.locs$ **do**

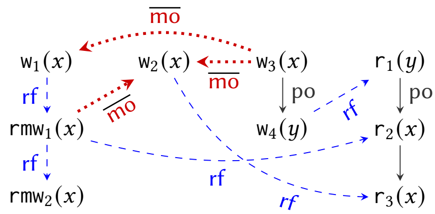
16     **if** $(\mathsf{rf}_x \cup \mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)$ *is cyclic* **then declare** 'Inconsistent'

17 **declare** 'Consistent'

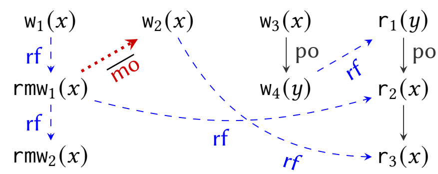
However, these witness relations are stronger than minimal coherence, while the algorithms for computing them (and thus checking consistency) have a higher polynomial complexity $O(n^3)$ (or $O(n^2 \cdot k)$) compared to our nearly linear bound.

On a more technical level, not every total extension of $(\mathsf{rf}_x \cup \mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)$ qualifies as the complete $\mathsf{mo}_x$ that witnesses consistency; in particular, some extensions might violate atomicity. This is also the case in prior witness relations [Abdulla et al. 2018; Lahav and Vafeiadis 2015; Luo and Demsky 2021]. However, a key difference between prior work and minimal coherence is the following. In prior witness relations, the events of an $\mathsf{rf}$-chain are either totally ordered or unordered with respect to any event outside this chain. In contrast, minimal coherence allows only some events of the $\mathsf{rf}$-chain being ordered with outside events. For example, in Fig. 8 observe that $(\mathtt{rmw}_1, \mathtt{w}_2) \in \overline{\mathsf{mo}}$. This implies that, due to atomicity, the pair $(\mathtt{rmw}_2, \mathtt{w}_2)$ must be ordered in any valid total extension of $\overline{\mathsf{mo}}$. However, minimal coherence does not force $(\mathtt{rmw}_2, \mathtt{w}_2)$ in $\overline{\mathsf{mo}}$. Nevertheless, our proof of Lemma 4.4 shows that, as long as $\overline{\mathsf{mo}}_x$ is minimally coherent, there always exists an extension $\overline{\mathsf{mo}}'_x \supseteq \overline{\mathsf{mo}}_x$ that can serve as the witnessing modification order, in the spirit of the prior notions of witness relations. In Fig. 8, for example, this extension would be $\overline{\mathsf{mo}}'_x = \overline{\mathsf{mo}}_x \cup (\mathtt{rmw}_2, \mathtt{w}_3)$.

**Algorithm.** The insights made above are turned into a consistency checking procedure in Algorithm 3. This algorithm first verifies the absence of $(\mathsf{po} \cup \mathsf{rf})$ cycles (which also implies that $\mathsf{hb} \subseteq (\mathsf{po} \cup \mathsf{rf})^+$ is irreflexive), and that $\mathsf{rf}$ follows weak-atomicity (Line 1). Then, it computes auxiliary data discussed in Section 3 (Lines 2-4). The main computation is performed in Lines 6-14, where the algorithm constructs a minimally coherent partial modification order $\overline{\mathsf{mo}}_x$ for each location $x$. The algorithm iterates over all read/RMW events $e$ accessing some location $x$, and identifies $\mathtt{w}_{\mathsf{rf}} = \mathsf{rf}^{-1}(e)$. Then, it iterates over all threads $u$ and identifies the $\mathsf{po}$-maximal write/RMW event $\mathtt{w}_u$ such that either $(\mathtt{w}', e) \in \mathsf{hb}$ (in which case $\mathtt{w}'$ is the event $\mathtt{w}^{\mathtt{w}}_u$ in Line 11) or $(\mathtt{w}', e) \in \mathsf{rf}; \mathsf{hb}$ (in which case $\mathtt{w}'$ is the event $\mathtt{w}^{\mathtt{r}}_u$ in Line 11). It then checks whether $(\mathtt{w}_u, \mathtt{w}_{\mathsf{rf}}) \notin \mathsf{rf}^+$, by checking that either $\mathtt{w}_u$ and $\mathtt{w}_{\mathsf{rf}}$ belong to different $\mathsf{rf}$-chains ('$(\mathbb{TC}[\mathtt{w}_{\mathsf{rf}}] \neq \mathbb{TC}[\mathtt{w}_u])$'), or $\mathtt{w}_{\mathsf{rf}}$ appears earlier than $\mathtt{w}_u$ in




(a) All accesses are REL/ACQ.




(b) All accesses are RLX.

Fig. 9. A minimally coherent $\overline{\text{mo}}$ computed by Algorithm 3 (a) and Algorithm 4 (b).

the common rf-chain ('$(\mathbb{PC}[\mathsf{w}_{\mathsf{rf}}] < \mathbb{PC}[\mathsf{w}_u])$'); see Line 13. If so, the algorithm inserts an ordering $(\mathsf{w}_u, \mathbb{TC}[\mathsf{w}_{\mathsf{rf}}])$ in $\overline{\text{mo}}_x$ (Line 14). Finally, Line 16 verifies that $\overline{\text{mo}}_x$ satisfies write-coherence. Fig. 9a displays the resulting $\overline{\text{mo}}$ computed by Algorithm 3 on a partial execution.

**Correctness and complexity.** Completeness follows from Lemma 3.2: every ordering inserted in $\overline{\text{mo}}_x$ is present in any modification order mo that witnesses the consistency of $\overline{\mathsf{X}}$, while the acyclicity check in Line 16 is necessary for write-coherence. Hence, if the algorithm returns "Inconsistent", $\overline{\mathsf{X}} \not\models$ RC20. The soundness comes from the fact that $\overline{\text{mo}}$ satisfies Item (1) of minimal coherence at the end of the loop of Line 6. Since all orderings inserted in $\overline{\text{mo}}$ are to the top of an rf-chain, Item (2) of minimal coherence is trivially satisfied at all times. Finally, if the acyclicity check in Line 16 passes, $\overline{\text{mo}}$ also satisfies Item (3) of minimal coherence. Thus, by Lemma 4.4, we have $\overline{\mathsf{X}} \models$ RC20.

The time spent in computing $\mathbb{HB}$, $(\mathbb{TC}, \mathbb{PC})$ and accessing the lists $\{\mathbb{WList}^u_{t,x}\}_{t,x,u}$ and $\{\mathbb{RList}^u_{t,x}\}_{t,x,u}$ is $O(n \cdot k)$ (Section 3). The number of orderings added in $\overline{\text{mo}}$ is $O(n \cdot k)$, taking $O(n \cdot k)$ total time. For each location $x \in \mathsf{E}.locs$, the acyclicity check in Line 16 can be performed in $O(n_x \cdot k)$ time, where $n_x = |\mathsf{W}_x \cup \mathsf{RMW}_x|$. For this, we construct a graph $G_x$ that consists of all events $(\mathsf{W}_x \cup \mathsf{RMW}_x)$ and $O(n_x \cdot k)$ edges. Given two events $e_1 = (t_1, x)$, $e_2 = (t_2, x)$ we have an edge $e_1 \to e_2$ in $G_x$ iff $(e_1, e_2) \in (\text{rf} \cup \overline{\text{mo}}_x)$ or $e_1 = \text{lastWriteBefore}(t_1, x, \mathbb{HB}[e_2][t_1] - 1)$. We then check for a cycle in $G_x$. Repeating this for all locations $x$, we obtain $O(n \cdot k)$ total time. We thus arrive at Theorem 1.1.

THEOREM 1.1. *Consistency checking for* RC20 *can be solved in $O(n \cdot k)$ time.*

### 4.5 Consistency Checking for Relaxed

We now turn our attention to the Relaxed fragment. As a strict subset of RC20 (where hb = po), consistency checking for this model can be performed in $O(n \cdot k)$ time by Theorem 1.1. Although this bound is nearly linear time, here we show that the Relaxed fragment enjoys a *truly* linear time consistency checking, independent of $k$ (Theorem 1.6). This improvement is based on two insights.

As rf edges do not induce any synchronization in this fragment, our first insight is that the input partial execution $\overline{\mathsf{X}} = (\mathsf{E}, \text{po}, \text{rf})$ can be partitioned into separate executions $\overline{\mathsf{X}}_x = (\mathsf{E}_x, \text{po}_x, \text{rf}_x)$, one for each location $x \in \mathsf{E}.locs$. Indeed, we have $\overline{\mathsf{X}} \models$ Relaxed iff $\text{acy}(\text{po} \cup \text{rf})$ and $\overline{\mathsf{X}}_x \models$ Relaxed for each $x \in \mathsf{E}.locs$. Thus, without loss of generality, we may assume that $\overline{\mathsf{X}}$ consists of a single location. Our second insight comes from the simplified formulation of minimal coherence under Relaxed.

**Minimal coherence under** Relaxed**.** Let us revisit the concept of minimal coherence under RC20. Focusing on the Relaxed fragment, we have hb = po. Thus, the first and third conditions of minimal coherence are reduced to the following.

(1′) For every triplet $(\mathsf{w}, \mathsf{r}, \mathsf{w}')$ accessing location $x$, if $(\mathsf{w}', \mathsf{r}) \in \text{rf}^?; \text{po}$ and $(\mathsf{w}', \mathsf{w}) \notin \text{rf}^+$, we have $(\mathsf{w}', \mathbb{TC}[\mathsf{w}]) \in (\text{rf}_x \cup \text{po}_x \cup \overline{\text{mo}}_x)^+$.
(3′) $(\text{rf}_x \cup \text{po}_x \cup \overline{\text{mo}}_x)$ is acyclic, for each $x \in \mathsf{E}.locs$.

**Algorithm 4:** Checking consistency for Relaxed.

**Input:** Events $\mathsf{E}$, program order $\mathsf{po}$ and reads-from relation $\mathsf{rf}$

1 **if** $(\mathsf{po} \cup \mathsf{rf})$ *is cyclic or* $\mathsf{rf}$ *violates weak-atomicity* **then declare** 'Inconsistent'
2 **let** $\mathbb{TC}$ and $\mathbb{PC}$ be $\mathsf{E}$-indexed arrays denoting the top and position of events in their $\mathsf{rf}$-chains
3 **foreach** $x \in \mathsf{E}.locs$ **do**
4     **foreach** $t \in \mathsf{E}_x.tids$ **do** $\mathsf{LW}_{t,x} \leftarrow \mathtt{NIL}$
5     $\overline{\mathsf{mo}}_x \leftarrow \emptyset$
6     **foreach** $e \in \mathsf{E}_x$ *in* $(\mathsf{po}_x \cup \mathsf{rf}_x)$*-order* **do**
7         **case** $e = \mathtt{w}(t, x)$ **do**
8             $\mathsf{LW}_{t,x} \leftarrow e$
9         **case** $e = \mathtt{r}(t, x)$ **do**
10             **if** $(\mathbb{TC}[\mathsf{rf}_x^{-1}(e)] \neq \mathbb{TC}[\mathsf{LW}_{t,x}])$ *or* $(\mathbb{PC}[\mathsf{rf}_x^{-1}(e)] < \mathbb{PC}[\mathsf{LW}_{t,x}])$ **then**
11                 $\overline{\mathsf{mo}}_x \leftarrow \overline{\mathsf{mo}}_x \cup \{(\mathsf{LW}_{t,x}, \mathbb{TC}[\mathsf{rf}_x^{-1}(e)])\}$
12             $\mathsf{LW}_{t,x} \leftarrow \mathsf{rf}^{-1}(e)$
13         **case** $e = \mathtt{rmw}(t, x)$ **do**
14             Execute Lines 10-11 followed by Line 8
15     **if** $(\mathsf{po}_x \cup \mathsf{rf}_x \cup \overline{\mathsf{mo}}_x)$ *is cyclic* **then declare** 'Inconsistent'
16 **declare** 'Consistent'

Similarly to RC20, Fig. 8 also serves as an illustration of the above definition. The key insight towards a truly linear-time algorithm is as follows. Consider the execution of Algorithm 3 on a partial execution $\overline{\mathsf{X}}$ under Relaxed semantics. Further, consider a read/RMW event $e$ processed by the algorithm with $\mathsf{w}_{\mathsf{rf}} = \mathsf{rf}^{-1}(e)$. Among all events $\mathsf{w}'$ forming a triplet $(\mathsf{w}_{\mathsf{rf}}, e, \mathsf{w}')$ and such that $(\mathsf{w}', e) \in \mathsf{rf}^?; \mathsf{po}$, there exists one that is $(\mathsf{rf} \cup \mathsf{po} \cup \overline{\mathsf{mo}}_x)^+$-maximal. In particular, if the immediate $\mathsf{po}$-predecessor of $e$ is a read event $\mathsf{r}$, then the event $\mathsf{rf}^{-1}(\mathsf{r})$ is this maximal $\mathsf{w}'$. Otherwise, the immediate $\mathsf{po}$-predecessor of $e$ is a write/RMW event $\mathsf{w}''$, which is also the maximal $\mathsf{w}'$. Thus, it suffices to keep track of this information on-the-fly, and only insert $(\mathsf{w}', \mathbb{TC}[\mathsf{w}_{\mathsf{rf}}])$ in $\overline{\mathsf{mo}}_x$, if necessary, to make $\overline{\mathsf{mo}}_x$ minimally-coherent. As we now do not have to compute $\mathbb{HB}$-timestamps or iterate over all threads during the processing of $e$, we have a truly linear-time algorithm.

**Algorithm.** The above insights are turned into an algorithm in Algorithm 4. The algorithm first verifies that $(\mathsf{po} \cup \mathsf{rf})$ is acyclic and $\mathsf{rf}$ satisfies weak-atomicity (Line 1). Then, it performs a separate pass for each location $x$ and constructs the minimally coherent $\overline{\mathsf{mo}}_x$. To this end, it keeps track in $\mathsf{LW}_{t,x}$ the unique $(\mathsf{rf}_x \cup \mathsf{po}_x \cup \overline{\mathsf{mo}}_x)^+$-maximal write/RMW event that has an $\mathsf{rf}^?; \mathsf{po}$ path to the current event of thread $t$. When a read/RMW event $e$ is processed, the algorithm potentially updates $\overline{\mathsf{mo}}_x$ with an ordering $(\mathsf{LW}_{t,x}, \mathbb{TC}[\mathsf{w}_{\mathsf{rf}}])$ (Line 10), using the same condition as in Algorithm 3. Fig. 9 contrasts the $\overline{\mathsf{mo}}$ computed by Algorithm 4 to the $\overline{\mathsf{mo}}$ computed by Algorithm 3 on the same partial execution but with different access levels. We arrive at the following theorem.

Theorem 1.6. *Consistency checking for* Relaxed *can be solved in* $O(n)$ *time.*

### 4.6 A Super-Linear Lower Bound for RMW-Free RA, WRA, and SRA

Finally, we address the existence of a truly linear-time algorithm for any model other than Relaxed. We show that this is unlikely, by proving the two lower bounds of Theorem 1.7. The proof is via a *fine-grained* reduction from the problem of checking triangle freeness in undirected graphs, which suffers the same lower bounds. That is, there is no algorithm (resp. combinatorial algorithm, under the BMM hypothesis) for checking triangle-freeness in time $O(n^{\omega/2-\epsilon})$ (resp. $O(n^{3/2-\epsilon})$) for any fixed $\epsilon > 0$ [Williams and Williams 2018], where $n$ is the number of nodes in the graph.

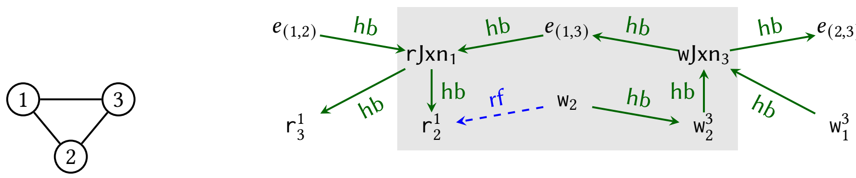



Fig. 10. *Left:* A graph $G$ with three nodes $V_G = \{1, 2, 3\}$ containing a triangle. *Right:* A slice of the partial execution $\overline{\mathsf{X}}$ for $G$. We have $(\mathsf{w}_2, \mathsf{w}^3_2) \in \mathsf{hb}$ and $(\mathsf{w}^3_2, \mathsf{r}^1_2) \in \mathsf{hb}$, thus violating weak read coherence in $\overline{\mathsf{X}}$.

**Reduction.** Given a graph $G = (V_G, E_G)$ of $n$ vertices, we construct an RMW-free partial execution $\overline{\mathsf{X}} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf} \rangle$ with $|\mathsf{E}| = O(n)$ such that $\overline{\mathsf{X}}$ is consistent with any of RA, WRA and SRA iff $G$ is triangle-free. For simplicity, we let $V_G = \{1, \dots, n\}$.

*Events and memory locations.* We start with the event set E. For the moment, all events belong to different threads, while we only define the memory location of an event when relevant. For every node $\alpha \in V_G$, $\overline{\mathsf{X}}$ contains (i) a location $y_\alpha$ and a write event $\mathsf{w}_\alpha(y_\alpha)$ and (ii) auxiliary "junction" events $\mathsf{rJxn}_\alpha$ and $\mathsf{wJxn}_\alpha$ on fresh locations. For every edge $(\alpha, \beta) \in E_G$ with $\alpha < \beta$, $\overline{\mathsf{X}}$ contains (i) an event $e_{(\alpha,\beta)}$ that accesses a fresh location, (ii) a read event $\mathsf{r}^\alpha_\beta(y_\beta)$, and (iii) a write event $\mathsf{w}^\beta_\alpha(y_\alpha)$.

*Relations* rf *and* hb. We now define the rf relation. Our construction also makes certain events hb ordered. This can be done trivially by introducing auxiliary events with an rf relation between them, while $\overline{\mathsf{X}}$ remains of size $O(n)$. In particular, every $(e_1, e_2) \in \mathsf{hb}$ edge can be simulated using fresh events $\mathsf{r}, \mathsf{w}$ such that (i) $(\mathsf{w}, \mathsf{r}) \in \mathsf{rf}$ (ii) $(e_1, \mathsf{w}) \in \mathsf{po}$, and (iii) $(\mathsf{r}, e_2) \in \mathsf{po}$. For simplicity of presentation, we do not mention these events explicitly, but rather directly the hb relation they result in. For every edge $(\alpha, \beta) \in E_G$ with $\alpha < \beta$, we have the following relations:

$$(\mathsf{w}_\beta, \mathsf{r}^\alpha_\beta) \in \mathsf{rf} \qquad (\mathsf{w}_\alpha, \mathsf{w}^\beta_\alpha) \in \mathsf{hb} \qquad (\mathsf{w}^\beta_\alpha, \mathsf{wJxn}_\beta) \in \mathsf{hb}$$
$$(\mathsf{wJxn}_\beta, \mathsf{e}_{(\alpha,\beta)}) \in \mathsf{hb} \qquad (\mathsf{e}_{(\alpha,\beta)}, \mathsf{rJxn}_\alpha) \in \mathsf{hb} \qquad (\mathsf{rJxn}_\alpha, \mathsf{r}^\alpha_\beta) \in \mathsf{hb}$$

Fig. 10 illustrates the above construction for a slice of the constructed partial execution $\overline{\mathsf{X}}$. We conclude with a proof sketch of Theorem 1.7, and refer to [Tunç et al. 2023a] for the full proof.

**Correctness and time complexity.** If there is a triangle $(\alpha, \beta, \gamma)$ in $G$ with $\alpha < \beta, \gamma$, then $(\mathsf{w}^\gamma_\beta, \mathsf{r}^\alpha_\beta) \in \mathsf{hb}$ because of the sequence of hb edges: $(\mathsf{w}^\gamma_\beta, \mathsf{wJxn}_\gamma)$, $(\mathsf{wJxn}_\gamma, \mathsf{e}_{(\alpha,\gamma)})$, $(e_{(\alpha,\gamma)}, \mathsf{rJxn}_\alpha)$, $(\mathsf{rJxn}_\alpha, \mathsf{r}^\alpha_\beta)$. Together with $(\mathsf{w}_\beta, \mathsf{r}^\alpha_\beta) \in \mathsf{rf}$ and $(\mathsf{w}_\beta, \mathsf{w}^\gamma_\beta) \in \mathsf{hb}$ by construction, we obtain a weak-read-coherence violation. In the other direction, if there are no triangles in $G$, then the modification order $\mathsf{mo} = \bigcup_{\alpha \in V_G} \mathsf{mo}_{y_\alpha}$ where $\mathsf{mo}_{y_\alpha}$ orders $\mathsf{w}_\alpha$ before every other write $\mathsf{w}^\beta_\alpha$ on $y_\alpha$, makes $\mathsf{X} = \langle \overline{\mathsf{X}}.\mathsf{E}, \overline{\mathsf{X}}.\mathsf{po}, \overline{\mathsf{X}}.\mathsf{rf}, \mathsf{mo} \rangle$ SRA- (and thus also RA- and WRA-) consistent. Such an mo ensures that $(\mathsf{hb} \cup \mathsf{mo})$ is acyclic. Triangle-freeness ensures read-coherence — a violation of read coherence implies that there are three events $e_1 = \mathsf{w}_\beta, e_2 = \mathsf{w}^\gamma_\beta, e_3 = \mathsf{r}^\alpha_\beta$ such that $(e_1, e_3) \in \mathsf{rf}$, $(e_1, e_2) \in \mathsf{mo}$ and $(e_2, e_3) \in \mathsf{hb}$, implying a triangle $(\alpha, \beta, \gamma)$ in $G$. The total time to construct $\overline{\mathsf{X}}$ is $O(|V_G| + |E_G|)$. Further, our reduction is completely combinatorial. We thus arrive at the following theorem.

Theorem 1.7. *There is no consistency-checking algorithm for the RMW-free fragments of any of* RA, WRA, *and* SRA *that runs in time $O(n^{\omega/2-\epsilon})$, for any fixed $\epsilon > 0$. Moreover, there is no combinatorial algorithm for the problem that runs in time $O(n^{3/2-\epsilon})$, under the combinatorial BMM hypothesis.*

## 5 EXPERIMENTAL EVALUATION

We implemented our consistency-checking algorithms for RA/RC20 and evaluated their performance on two standard settings of program analysis, namely, (i) *stateless model checking*, using

TruSt [Kokologiannakis et al. 2022], and (ii) *online testing*, using C11Tester [Luo and Demsky 2021]. These tools are designed to handle variants of C11 including SC accesses, and performing race-detection, which are beyond the scope of this work. Here, we focus on the consistency-checking component for the RA/RC20 fragment, which is common in these tools and our work. We conducted our experiments on a machine running Ubuntu 22.04 with 2.4GHz CPU and 64GB of memory.

**Benchmarks.** We used standard benchmark programs from prior state-of-the-art verification and testing papers [Abdulla et al. 2018; Kokologiannakis and Vafeiadis 2021; Luo and Demsky 2021; Norris and Demsky 2013], as well as the applications Silo, GDAX, Mabain, and Iris [Luo and Demsky 2021] for online testing. These benchmarks use C11 concurrency primitives extensively. For thorough evaluation, we have scaled up some of them, when their baseline versions were too small, by increasing the number of threads or loop counters. Some benchmarks also contain accesses outside our scope; we converted those accesses to access modes applicable for our experiments, in line with the evaluation in prior works [Abdulla et al. 2018; Lahav and Margalit 2019].

### 5.1 Stateless Model Checking

The TruSt model checker explores all behaviors of a bounded program by enumerating executions, making use of different strategies to avoid redundant exploration. One such strategy is to enumerate partial executions $\overline{\mathsf{X}}$ and perform a consistency check for the maximal ones, to verify that they represent valid program behavior. As the number of explored executions is typically large, it is imperative that consistency checks are performed as fast as possible.

**Consistency checking inside TruSt.** The algorithm for consistency checking in TruSt constructs a writes-before order wb [Lahav and Vafeiadis 2015], which is a partial modification order that serves as a witness of consistency. The time taken to construct wb is $O(n^3)$, which has been identified as a bottleneck in the model checking task [Kokologiannakis et al. 2022, 2019]. We replaced TruSt's wb algorithm for consistency checking with the $\overline{\mathsf{mo}}$ computation of Algorithm 3, and measured (i) the speedup realized for consistency checking, and (ii) the effect of this speedup on the overall model-checking task. We executed TruSt on several benchmarks, each with a time budget of 2 hours, measuring the average time for consistency checking (for evaluating (i)) as well as the total number of executions explored (for evaluating (ii)). Finally, we note that TruSt employs a number of simpler consistency checks during the exploration. Although we expect that our new algorithm can improve those as well, we have left them intact as it was unclear to us how they interact with the rest of the tool, and in order to maintain soundness of the obtained results.

**Experimental results.** Our results are shown in Table 1. We mark with † benchmarks on which the model checker found an error and halted early. We observe that Algorithm 3 is always faster, typically by a significant margin. The maximum speedup for consistency checking is 162×, and the geometric mean of speedups is 36×. Regarding the number of executions, the model checker explores 4.3× more on (geometric) average, and as high as 71.6× more, when using Algorithm 3. In some benchmarks, the two approaches observe a similar number of executions. This is due to consistency checking being only part of the overall model-checking procedure, which also consists of other computationally intensive tasks such as backtracking. As consistency checking appears now to not be a bottleneck, it is meaningful to focus further optimization efforts on these other tasks. For ttaslock, we noticed a livelock that blocks the model checker. Overall, our experiments highlight that the benefit of the new, nearly linear time property of consistency checking leads to a measurable speedup that positively impacts the overall efficiency of model checking. We refer to [Tunç et al. 2023a] for experiments on RC20, which lead to the same qualitative conclusions.

Table 1. Impact on model checking. Columns 2 and 3 denote the average time (in seconds) spent in consistency checking by resp. TruSt and our algorithm. Columns 5 and 6 denote the total number of executions explored by resp. TruSt and our algorithm. Columns 4 and 7 denote the respective speedups and ratios.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Avg. Time** | | | **Executions** | | | **Benchmark** | **Avg. Time** | | | **Executions** | | |
| | **TruSt** | **Our Alg.** | **S** | **TruSt** | **Our Alg.** | **R** | | **TruSt** | **Our Alg.** | **S** | **TruSt** | **Our Alg.** | **R** |
| barrier | 0.5 | 0.01 | 45.0 | 14K | 81K | 5.98 | ms-queue† | 0.07 | 0.01 | 7.0 | 208 | 208 | 1 |
| buf-ring | 1.6 | 0.02 | 79.5 | 3K | 6K | 2.52 | mutex | 0.8 | 0.01 | 78.0 | 7K | 218K | 29.67 |
| chase-lev† | 0.0 | 0.0 | - | 2 | 2 | 1 | peterson | 0.3 | 0.01 | 26.0 | 4K | 5K | 1.16 |
| control-flow | 0.08 | 0.01 | 8.0 | 88K | 1M | 16.56 | qu | 0.1 | 0.01 | 12.0 | 2K | 2K | 1.04 |
| dekker | 0.2 | 0.01 | 24.0 | 26K | 64K | 2.45 | seqlock | 0.2 | 0.01 | 24.0 | 24K | 171K | 7.02 |
| dq | 0.1 | 0.01 | 13.0 | 40K | 163K | 4.11 | sigma | 0.2 | 0.01 | 22.0 | 4K | 62K | 5.97 |
| exp-bug | 0.1 | 0.01 | 13.0 | 55K | 2M | 29.98 | spinlock | 1.0 | 0.01 | 98.0 | 5K | 31K | 6.04 |
| fib-bench | 1.6 | 0.01 | 162.0 | 4K | 315K | 71.63 | szymanski | 0.4 | 0.01 | 44.0 | 2K | 2K | 1.16 |
| gcd | 0.3 | 0.01 | 31.0 | 18K | 80K | 4.46 | ticketlock | 0.9 | 0.02 | 45.5 | 7K | 89K | 12.47 |
| lamport | 0.4 | 0.01 | 36.0 | 17K | 93K | 5.54 | treiber | 0.6 | 0.01 | 58.0 | 403 | 403 | 1 |
| linuxrwlocks | 0.4 | 0.01 | 40.0 | 12K | 32K | 2.75 | ttaslock | 1.0 | 0.01 | 97.0 | 401 | 401 | 1 |
| mcs-spinlock | 1.1 | 0.01 | 108.0 | 5K | 38K | 7.19 | twalock | 0.5 | 0.01 | 52.0 | 8K | 30K | 3.58 |
| mpmc | 0.7 | 0.01 | 66.0 | 10K | 77K | 8.01 | | | | | | | |
| **Totals** | - | - | - | - | - | - | - | **14.5** | **0.26** | - | **356K** | **4.6M** | - |

## 5.2 Online Testing

We now turn our attention to the online testing setting using C11Tester's framework. In C11Tester's setting, a partial execution $\overline{\mathsf{X}}$ is constructed incrementally, by iteratively (i) revealing a randomly chosen new read/RMW event $\mathsf{r}$, (ii) choosing a valid writer $\mathsf{rf}(\mathsf{r})$, and (iii) continuing the execution of the program until the next read/RMW events. Hence, every iteration requires a consistency check. Although we could use Algorithm 3 from scratch at each step, this would result in unnecessary recomputations of $\overline{\mathsf{mo}}$. Instead, we follow a different approach here — we maintain $\overline{\mathsf{mo}}$ on-the-fly, in a way that incremental consistency checks can be done more efficiently.

**Incremental consistency checking.** Our incremental algorithm constructs a similar minimally coherent partial modification order $\overline{\mathsf{mo}}$ as our offline algorithm (Algorithm 3). However, unlike the offline setting, we need efficient incremental consistency checks. For this, we maintain a per-location order $\mathsf{hbmo}_x$ on write/RMW events that satisfies following invariants: (i) $\mathsf{hbmo}_x \subseteq (\mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)^+$ and (ii) $(\mathsf{hb}_x \cup (\overline{\mathsf{mo}}_x; \mathsf{po}_x^?)) \subseteq \mathsf{hbmo}_x$. In order to decide whether a new read/RMW event $\mathsf{r}(x)$ can observe a write/RMW event $\mathsf{w}(x)$, we must determine if there exists another write/RMW event $\mathsf{w}'(x)$ such that $(\mathsf{w}', \mathsf{r}) \in \mathsf{hb}_x$ and $(\mathsf{w}', \mathsf{w}) \in (\mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)^+$, as this would lead to a consistency violation. Using $\mathsf{hbmo}_x$, this check is performed as follows: (a) for each thread $u$, we identify the $\mathsf{po}$-maximal write/RMW event $\mathsf{w}'(u, x)$ for which $(\mathsf{w}', \mathsf{r}) \in \mathsf{hb}$, and (b) we update $\mathsf{hbmo}_x \leftarrow \mathsf{hbmo}_x \cup \{(\mathsf{w}'', \mathsf{w}') \mid (\mathsf{w}'', \mathsf{w}') \in \mathsf{hbmo}_x^+\}$. Due to invariant (ii), at this point we are guaranteed that, for all write events $\mathsf{w}''$, we have $(\mathsf{w}'', \mathsf{w}') \in \mathsf{hbmo}_x$ iff $(\mathsf{w}'', \mathsf{w}') \in (\mathsf{hb}_x \cup \overline{\mathsf{mo}}_x)^+$. We can now test whether $\mathsf{w}$ is a valid writer for $\mathsf{r}$ by checking whether $(\mathsf{w}, \mathsf{w}') \in \mathsf{hbmo}_x$, for one of the aforementioned write/RMW events $\mathsf{w}'$. We refer to [Tunç et al. 2023a] for implementation details.

**Main differences with C11Tester.** The consistency-checking algorithm implemented inside C11Tester also infers $\mathsf{mo}$ orderings as implied by read and write coherence. The two key differences between that approach and our incremental algorithm described above are the following: (i) C11Tester's $\mathsf{mo}$ is stronger than our minimally coherent $\overline{\mathsf{mo}}$ that is contained in $\mathsf{hbmo}$, and (ii) this $\mathsf{mo}$ is always maintained transitively-closed. These two differences are expected to make hbmo computationally cheaper to maintain than C11Tester's mo. Although we also have to compute transitive paths hbmo$^+$ when encountering read/RMW operations (step (b) above), in our experience, these paths typically touch a small part of the input, leading to an efficient computation.

Table 2. Impact on online testing. Columns 2, 3 and 4 give the average number of events, threads and locations in each benchmark. Columns 5 and 6 denote the average times in seconds to check for consistency by resp. C11Tester and our algorithm. Column 6 denotes the speedup.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **n** | **k** | **d** | **C11Test.** | **Our Alg.** | **SpeedUp** | **Benchmark** | **n** | **k** | **d** | **C11Test.** | **Our Alg.** | **SpeedUp** |
| control-flow | 52K | 25 | 3 | 4.2 | 0.04 | 104.25 | mutex | 15M | 11 | 11 | 20.9 | 16.8 | 1.24 |
| sigma | 36K | 10 | 9 | 2.4 | 0.03 | 80 | gdax | 11M | 5 | 46K | 7.0 | 5.7 | 1.23 |
| dq | 599K | 4 | 2 | 22.1 | 0.5 | 46.78 | spinlock | 5M | 11 | 10 | 7.0 | 5.9 | 1.18 |
| iris-1 | 1M | 12 | 45 | 28.4 | 1.8 | 16.25 | ticketlock | 14M | 6 | 20 | 15.7 | 13.2 | 1.18 |
| seqlock | 478K | 17 | 20 | 14.4 | 1.4 | 10.38 | ttaslock | 5M | 11 | 10 | 7.8 | 6.6 | 1.18 |
| exp-bug | 2M | 4 | 2 | 1.6 | 0.7 | 2.35 | fib-bench | 6M | 3 | 2 | 3.3 | 2.8 | 1.18 |
| chase-lev | 7M | 5 | 2 | 12.4 | 5.4 | 2.3 | qu | 1M | 10 | 29 | 1.6 | 1.3 | 1.18 |
| linuxrwlocks | 7M | 6 | 10 | 10.9 | 5.9 | 1.84 | treiber | 1M | 6 | 11 | 1.1 | 1.0 | 1.12 |
| mabain | 5M | 6 | 18 | 7.3 | 4.0 | 1.82 | silo | 8M | 4 | 4K | 5.3 | 4.8 | 1.1 |
| iris-2 | 12M | 3 | 12 | 9.9 | 5.7 | 1.72 | barrier | 8M | 5 | 20 | 7.8 | 7.4 | 1.04 |
| mcs-lock | 10M | 11 | 30 | 17.8 | 12.2 | 1.45 | mpmc | 9M | 10 | 3 | 12.9 | 12.5 | 1.03 |
| lamport | 6M | 3 | 5 | 3.4 | 2.5 | 1.36 | indexer | 2M | 17 | 128 | 1.6 | 1.5 | 1.03 |
| peterson | 5M | 3 | 4 | 2.9 | 2.2 | 1.29 | buf-ring | 5M | 9 | 12 | 7.0 | 6.8 | 1.02 |
| spsc | 10M | 3 | 699 | 6.4 | 5.0 | 1.28 | ms-queue | 4M | 11 | 13 | 8.2 | 8.2 | 0.99 |
| dekker | 16M | 3 | 3 | 9.2 | 7.2 | 1.27 | gcd | 5M | 3 | 2 | 2.2 | 2.5 | 0.89 |
| twalock | 4M | 11 | 4K | 8.9 | 7.0 | 1.26 | szymanski | 4M | 3 | 3 | 1.8 | 2.3 | 0.81 |
| **Totals** | - | - | - | - | - | - | - | **196M** | - | - | **286.4** | **170.8** | - |

**Experimental results.** Our results are shown in Table 2. For robust measurements, we report averages over 10 executions per benchmark, focusing on benchmarks for which at least one algorithm took $\geq$ 1s. Our approach achieves a maximum speedup of 104.2× and a geometric speed-up of 2×. In more detail, we observe significant improvement in the first 5 benchmarks, and consistent speedups of at least 1.2× on 18 benchmarks. We have encountered only 2 benchmarks, gcd and szymanski, on which the new algorithm is arguably slower. These benchmarks contain no RMWs and only a small number of write events. This results in the computation of very small modification orders, diminishing the benefit of our algorithm and results in a marginal slowdown.

## 6 CONCLUSION

Checking the reads-from consistency of concurrent executions is a fundamental computational task in the development of formal concurrency semantics, program verification and testing. In this paper we have addressed this problem in the context of C11-style weak memory models, for which this problem is both highly meaningful, and intricate. We have developed a collection of algorithms and complexity results that are either optimal or nearly-optimal, and thus accurately characterize the complexity of the problem in this setting. Further, our experimental evaluation indicates that the new algorithms have a measurable, and often significant, impact on the consistency-checking tasks that arise in practice. Thus our algorithms enable the development of more performant and scalable program analysis tools in this domain. This work is focused on non-SC fragments of C11, as otherwise, consistency checking inherits the $\mathcal{NP}$-hardness of SC consistency checking. For applications having an abundance of SC accesses, however, a meaningful direction for future work is to combine our techniques with heuristics developed for checking SC consistency (e.g., [Abdulla et al. 2018; Pavlogiannis 2019]), and apply them on programs that mix all types of C11 accesses.

## ACKNOWLEDGMENTS

Andreas Pavlogiannis was partially supported by a research grant (VIL42117) from VILLUM FONDEN. Umang Mathur was partially supported by the Simons Institute for the Theory of Computing, and by a Singapore Ministry of Education (MoE) Academic Research Fund (AcRF) Tier 1 grant. Shankaranarayanan Krishna was partially supported by the SERB MATRICS grant MTR/2019/000095. Parosh Aziz Abdulla was partially supported by the Swedish Research Council.

## DATA AND SOFTWARE AVAILABILITY STATEMENT

The artifact developed for this work is available [Tunç et al. 2023b], which contains all source codes and experimental data necessary to reproduce our evaluation in Section 5.

## REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 150:1–150:29. https://doi.org/10.1145/3360576

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

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 341–366. https://doi.org/10.1007/978-3-030-81685-8_16

Jade Alglave. 2010. *A Shared Memory Poetics.* Ph. D. Dissertation. l'Université Paris 7 - Denis Diderot.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

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

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 634–648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

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

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The Reads-from Equivalence for the TSO and PSO Memory Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 164 (2021), 30 pages. https://doi.org/10.1145/3485541

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends® in Programming Languages* 1, 1-2 (2014), 1–150. https://doi.org/10.1561/2500000011

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

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (2017), 30 pages. https://doi.org/10.1145/3158119

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 124 (2019), 29 pages. https://doi.org/10.1145/3360550

Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. 2004. Linear FPT Reductions and Computational Lower Bounds. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (Chicago, IL, USA) *(STOC '04)*. Association for Computing Machinery, New York, NY, USA, 212–221. https://doi.org/10.1145/1007352.1007391

Yunji Chen, Yi Lv, Weiwu Hu, Tianshi Chen, Haihua Shen, Pengyu Wang, and Hong Pan. 2009. Fast complete memory consistency verification. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 381–392. https://doi.org/10.1109/HPCA.2009.4798276

Peter Chini and Prakash Saivasan. 2020. A Framework for Consistency Algorithms. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 182)*, Nitin Saxena and Sunil Simon (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 42:1–42:17. https://doi.org/10.4230/LIPIcs.FSTTCS.2020.42

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

Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. 2011. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2011), 375–382. https://doi.org/10.1109/TPDS.2011.159

Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. 2015. Memory-Model-Aware Testing: A Unified Complexity Analysis. *ACM Trans. Embed. Comput. Syst.* 14, 4 (2015). https://doi.org/10.1145/2753761

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

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

ISO/IEC 9899. 2011. Programming Language C.

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (2018), 29 pages. https://doi.org/10.1145/3276516

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL, Article 19 (2023), 29 pages. https://doi.org/10.1145/3571212

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 6, POPL (2022). https://doi.org/10.1145/3498711

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification*. Springer-Verlag, Berlin, Heidelberg, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20

Ori Lahav and Udi Boker. 2022. What's Decidable About Causally Consistent Shared Memory? *ACM Trans. Program. Lang. Syst.* 44, 2, Article 8 (2022), 55 pages. https://doi.org/10.1145/3505273

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Roy Margalit. 2019. Robustness against Release/Acquire Semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 126–141. https://doi.org/10.1145/3314221.3314604

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. https://doi.org/10.

1145/3062341.3062352

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

Carl Lerche. 2020. Loom. Available at https://github.com/tokio-rs/loom.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

C. Manovit and S. Hangal. 2006. Completely verifying memory consistency of test program executions. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*. 166–175. https://doi.org/10.1109/HPCA.2006.1598123

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

Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (2018), 29 pages. https://doi.org/10.1145/3276515

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (2021), 29 pages. https://doi.org/10.1145/3434317

Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 530–545. https://doi.org/10.1145/3445814.3446748

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (2019), 29 pages. https://doi.org/10.1145/3371085

S. Qadeer. 2003. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems* 14, 8 (2003), 730–741. https://doi.org/10.1109/TPDS.2003.1225053

Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023a. Optimal Reads-From Consistency Checking for C11-Style Memory Models. arXiv:2304.03714

Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023b. *Optimal Reads-From Consistency Checking for C11-Style Memory Models*. https://doi.org/10.5281/zenodo.7816526 Artifact.

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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838

Virginia Vassilevska Williams. 2019. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians*. 3447–3487. https://doi.org/10.1142/9789813272880_0188

Virginia Vassilevska Williams and R. Ryan Williams. 2018. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *J. ACM* 65, 5, Article 27 (2018), 38 pages. https://doi.org/10.1145/3186893

Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2022. High-coverage metamorphic testing of concurrency support in C compilers. *Software Testing, Verification and Reliability* 32, 4 (2022). https://doi.org/10.1002/stvr.1812

Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2019. Gradual Consistency Checking. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 267–285. https://doi.org/10.1007/978-3-030-25543-5_16

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