Dynamic Robustness Verification against Weak Memory

ROY MARGALIT, Tel Aviv University, Israel
MICHALIS KOKOLOGIANNAKIS, ETH Zurich, Switzerland
SHACHAR ITZHAKY, Technion, Israel
ORI LAHAV, Tel Aviv University, Israel

Dynamic race detection is a highly effective runtime verification technique for identifying data races by instrumenting and monitoring concurrent program runs. However, standard dynamic race detection is incompatible with practical weak memory models; the added instrumentation introduces extra synchronization, which masks weakly consistent behaviors and inherently misses certain data races. In response, we propose to dynamically verify *program robustness*—a property ensuring that a program exhibits only strongly consistent behaviors. Building on an existing static decision procedures, we develop an algorithm for dynamic robustness verification under a C11-style memory model. The algorithm is based on "location clocks", a variant of vector clocks used in standard race detection. It allows effective and easy-to-apply defense against weak memory on a per-program basis, which can be combined with race detection that assumes strong consistency. We implement our algorithm in a tool, called RSan, and evaluate it across various settings. To our knowledge, this work is the first to propose and develop dynamic verification of robustness against weak memory models.

CCS Concepts: • **Software and its engineering** → *Compilers*; **Dynamic analysis**; **Concurrent programming languages**; *Formal software verification*; • **Theory of computation** → **Concurrency**; *Program verification.*

Additional Key Words and Phrases: Weak memory models, C/C++11, Robustness, Dynamic race detection

ACM Reference Format:
Roy Margalit, Michalis Kokologiannakis, Shachar Itzhaky, and Ori Lahav. 2025. Dynamic Robustness Verification against Weak Memory. *Proc. ACM Program. Lang.* 9, PLDI, Article 174 (June 2025), 24 pages. https://doi.org/10.1145/3729277

1 Introduction

Developers are well-aware that concurrent code is extremely hard to get right. With static verification methods not scaling to real-world programs, dynamic techniques have gained widespread success and adoption, and tools like ThreadSanitizer (TSan) have become essential parts of mainstream compilers [Serebryany and Iskhodzhanov 2009; ThreadSanitizer 2020]. Such tools detect *data races*—situations where two threads concurrently access the same variable, and at least one of them is writing—commonly leading to bugs in concurrent programs. To do so, they instrument the program to track synchronization at runtime and use the instrumentation for detecting races in the current run and for predicting races that will occur in other runs [Mathur and Pavlogiannis 2022].

Seminal work on dynamic race detection assumes that inter-thread synchronization is achieved using locks or strong synchronization variables (e.g., Java's volatile accesses) [Bond et al. 2010; Flanagan and Freund 2009; Savage et al. 1997]. More specifically, race detectors allow concurrent accesses to locks and synchronization variables, but consider races on all other variables as bugs,

Authors' Contact Information: Roy Margalit, Tel Aviv University, Tel Aviv, Israel, rm@tauex.tau.ac.il; Michalis Kokologiannakis, ETH Zurich, Zurich, Switzerland, michalis.kokologiannakis@inf.ethz.ch; Shachar Itzhaky, Technion, Haifa, Israel, shachari@cs.technion.ac.il; Ori Lahav, Tel Aviv University, Tel Aviv, Israel, orilahav@tau.ac.il.

ACM 2475-1421/2025/6-ART174
https://doi.org/10.1145/3729277

and report them to the user. Crucially, they assume that the synchronization variables follow *sequential consistency* (SC), i.e., accesses to these variables behave as if they are interleaved, and every read obtains its value from the latest write to the same variable.

In practice, however, synchronization variables provide guarantees weaker than SC. In particular, since 2011, C/C++ has introduced performance-oriented specialized synchronization accesses, known as *atomic accesses*, with several different strengths that allow behaviors weaker than SC, as well as *fences* for fine-grained control on the synchronization between these accesses. The semantics of atomic accesses and fences is defined by the C11 *weak (or relaxed) memory model* [Batty et al. 2011; ISO/IEC 14882:2011 2011; ISO/IEC 9899:2011 2011; Lahav et al. 2017].

In the context of a weak memory model like C11, dynamic race detection is especially valuable. Indeed, understanding the model and correctly synchronizing C11 code has been proven a difficult task, and developers are often advised to forgo the potential performance benefits of C11, and instead rely on the "good old" SC accesses and locks [Boehm and Adve 2008]. In turn, dynamic race detection would automatically identify subtle issues that lead to data races, and guide programmers in strengthening synchronization variables or adding fences to ensure sufficient synchronization.

Unfortunately, developing dynamic race detectors for languages with weak memory models is highly challenging due to what we term the "observer effect": the instrumentation code that monitors for data races requires its own synchronization mechanisms to avoid internal data races, which inadvertently masks non-SC behaviors of the original program. We demonstrate the consequences of the observer effect with the following example.

Example 1.1. For the program on the right, the behavior where both reads read 0 is forbidden under SC, but allowed by C11—and observable on standard laptops—if we use weakly consistent atomic accesses. However, as soon as we use a dynamic race detector like TSan on this program, the annotated outcome no longer manifests. While it is known that TSan does not fully handle the C11 weak memory model (including soundness issues[1]), it is perhaps less recognized that running TSan on a program can mask certain behaviors that the program might otherwise exhibit. A real-world case where TSan misses a race due to this problem, is the well-known (broken) implementation of Dekker's mutual exclusion protocol [Dijkstra 1965] using C11 release/acquire atomics. When run with TSan, the race within the intended critical section is never revealed.

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

Weak behaviors like the one above remain hidden during monitoring, but they will resurface in production code, as soon as the instrumentation is removed. In fact, even logging the executed accesses for a "post-mortem" analysis introduces extra synchronization: accessing the shared log requires a memory barrier, which again restricts the behaviors a program would otherwise exhibit.

To address this issue, one must essentially execute the program while simulating the C11 constraints. For instance, a read can no longer simply retrieve the most recent written value, but rather the instrumentation should (randomly) select a value of some write permitted to read-from by the memory model. Such an approach leads to a clear trade-off between completeness—i.e., potentially detecting *all* racy behaviors—and performance/memory overhead compared to standard execution. Existing methods either significantly sacrifice completeness [Lidbury and Donaldson 2017] or incur high memory overhead proportional to the execution length [Luo and Demsky 2021].

In this paper, we propose a novel approach to the problem of dynamic verification of concurrent programs under a weak memory model. Our key idea is to accompany dynamic race detection that assumes SC with a dynamic verification of *robustness*. Robustness is a general correctness

[1]See, e.g., https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97868 and https://github.com/google/sanitizers/issues/1415 (accessed November 2024)

criterion for concurrent programs under weak memory models that requires that all behaviors allowed under the weak model are also allowed under SC. It guarantees that verification in general, and dynamic race detection in particular, may ignore weak memory behaviors and work under the illusion of SC. Robustness does not rule out the use of weakly consistent atomics. Indeed, in a variety of cases, one can use weak accesses without making weak behaviors externally observable, thereby obtaining the best of both worlds: SC semantics and efficient implementations.

While a large body of previous work (see §7) has developed verification methods and tools for robustness, these methods are static, and do not scale to real-world full-blown programs. The static methods, however, reveal an important benefit of robustness, compared to other correctness notions: robustness is reducible to the absence of a certain shape *in executions under SC*. This property makes robustness an ideal candidate for dynamic verification, as robustness (and non-robustness) can be checked while ignoring weak memory behaviors, and is thus immune to the "observer effect".

Based on the above observation, we develop the first dynamic verification method for program robustness against a C11-style memory model (we target RC20, a variant of C11 from [Margalit and Lahav 2021], see §4.2). This means that we run the program, and check that no atomic access in the current run may exhibit a weak behavior according to the memory model, i.e., a load reading a stale value or a write becoming stale although it is executed last. Then, we accompany our technique with a dynamic race detector that assumes SC for detecting data races on *non-atomic* accesses. Dealing with a language-level model, we note that non-robustness does not imply the occurrence of weak behaviors, but rather signifies that such behaviors are permitted by the formal model and can potentially manifest when using specific compilers on particular architectures.

To dynamically verify robustness, we enhance the instrumentation used by race detectors to detect *non-robustness witnesses*. The latter, defined in Lahav and Margalit [2019]; Margalit and Lahav [2021], form our analog of data races. However, the instrumentation used in these works for *static* detection of non-robustness witnesses has significant drawbacks when used *dynamically* (see §3.1). To solve this, we develop an instrumentation based on *location clocks*, an adaptation of *vector clocks* [Fidge 1988; Mattern 1989] that assigns timestamps to memory locations, rather than threads. The instrumentation based on location clocks is better suited for dynamic analysis. Specifically, it reduces contention on locks for maintaining the instrumentation, and thus retains more of the concurrency in the input program. Moreover, it enables a *predictive* analysis that can identify non-robustness witnesses by observing executions that do not directly serve as such witnesses. The proposed technique is sound (i.e., never reports spurious robustness violations) and complete (i.e., if a program is not robust, there exists at least one schedule under which the analysis reports a violation), and its memory consumption is practically independent of the length of the run.

We implement our approach on top of TSan in a tool called RSan (Robustness Sanitizer), and evaluate it on multiple examples and case studies. RSan scales to real-world codebases with thousands of lines of code. Upon detecting a robustness violation, RSan reports the offending instructions to the user, who can then prevent the corresponding weak behavior by strengthening some accesses and/or inserting fences. With its simplicity of use and applicability to C/C++ as-is, we envision RSan being used in a defensive programming style, enabling developers to improve performance of their code without being experts in weak memory models.

Finally, we also develop a race detection algorithm for C/C++11 (specifically, its RC20 variant), and implement it in TSan's infrastructure. Although robustness (for atomics) permits the race detector to assume SC semantics, the detection algorithm must still account for the model's definition of data races on non-atomics, which are determined by the way atomics and fences are used.

The rest of the paper is organized as follows:

§2 We recap (a fragment of) C11 and its robustness verification technique from prior work.

§3 We introduce our dynamic-analysis algorithm based on location clocks.
§4 We present extensions to the basic algorithm: covering the full RC20 model and a technique to avoid reporting common benign robustness violations.
§5,§6 We discuss the implementation of our approach and evaluate our implementation.
§7 We discuss related work and outline potential future directions.

Appendices with the complete location-clock algorithm and the modified race detection algorithm are included in [Margalit et al. 2025]. The RSan tool and the examples used for its evaluation are available in the accompanying artifact available at https://doi.org/10.5281/zenodo.15002567.

2 Preliminaries

We begin by providing the necessary background on the C11 model (§2.1), the robustness criterion (§2.2), and the static verification approach previously proposed for robustness (§2.3). Figure 1 presents simple illustrative examples, which we reuse throughout this section.

Atomic accesses in C11 have memory orderings ("relaxed", "release", "acquire", and "sequentially consistent"), with the weak orderings being better-performant on multicore hardware compared to sequentially consistent ones. To simplify the presentation, in this section we only discuss *release/acquire write and read accesses*, a well-studied fragment of C11 [Lahav et al. 2016]. Our full development and the accompanying tool support a broader subset of C11, which we discuss in §4.

2.1 The Release/Acquire Memory Model

The C11 model is declarative, capturing program behavior in *execution graphs*—structures that track several partial orders on memory accesses, using them to limit the allowed return values of reads. Nodes in these graphs are called *events*, and correspond to memory accesses. Edges denote various relations between the accesses. Let us now formalize these notions.

Domains. We let $\mathsf{Loc} = \{\mathsf{x}, \mathsf{y}, \ldots\}$ be a finite set of locations, Val be a set of values that contains 0, which we use as the initial value, and $\mathsf{Tid} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots\}$ be a finite set of thread identifiers. In our code snippets, threads are numbered from left to right, with the leftmost thread labeled T_1.

Events. An *event* e is a tuple $\langle \tau, s, l \rangle$, where $\tau \in \mathsf{Tid} \uplus \{\bot\}$ is a thread identifier (or $\bot$ for initialization events), $s \in \mathbb{N}$ is a serial number inside each thread, and $l \in \mathsf{Lab}$ is a label, which can be either $\mathtt{R}(x, v)$ (read) or $\mathtt{W}(x, v)$ (write) with $x \in \mathsf{Loc}$ and $v \in \mathsf{Val}$. The functions $\mathtt{tid}$, $\mathtt{loc}$, and $\mathtt{val}$ return the thread identifier, location, and value of a given event. We let R and W denote the sets of all read and write events, respectively. We employ sub- and superscripts to restrict sets of events, e.g., $\mathsf{W}_x = \{w \in \mathsf{W} \mid \mathtt{loc}(w) = x\}$ and $E^\tau = \{e \in E \mid \mathtt{tid}(e) = \tau\}$ for a set E of events.

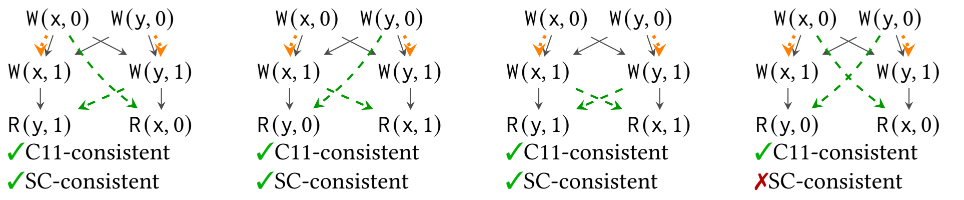
Execution Graphs. An *execution* graph G is a tuple $\langle E, rf, mo \rangle$, where:

- E is a finite set of events such that $\mathsf{Init} \subseteq E$, where $\mathsf{Init} \triangleq \{\langle \bot, 0, \mathtt{W}(x, 0) \rangle \mid x \in \mathsf{Loc}\}$ contains an initial write per location. We require that $\mathtt{tid}(e) \neq \bot$ for every $e \in E \setminus \mathsf{Init}$, and that no two different events in E have the same thread identifier and serial number.
- rf is a "reads-from" mapping that determines the write event from which each read reads its value. Formally, the following should hold:
  - If $\langle w, r \rangle \in rf$, then $w \in E \cap \mathsf{W}$, $r \in E \cap \mathsf{R}$, $\mathtt{loc}(w) = \mathtt{loc}(r)$, and $\mathtt{val}(w) = \mathtt{val}(r)$.
  - For every $r \in E \cap \mathsf{R}$, there exists exactly one write event w such that $\langle w, r \rangle \in rf$.
- mo is a "modification order" that totally orders the writes to each location. Formally, mo is a disjoint union of relations $\{mo_x\}_{x \in \mathsf{Loc}}$, such that each mo_x is a strict total order on $E \cap \mathsf{W}_x$.

We denote the components of G by $G.\mathtt{E}$, $G.\mathtt{rf}$, and $G.\mathtt{mo}$. We use $G.\mathsf{po}$ ("program order") to denote the (partial) order on $G.\mathtt{E}$ in which initialization events (with $\mathtt{tid}(e) = \bot$) precede all other events, Consider the following two litmus tests, where x, y, ... denote global memory locations, and a, b, ... local "registers". In both tests, we assume that all accesses are annotated as release/acquire atomics.

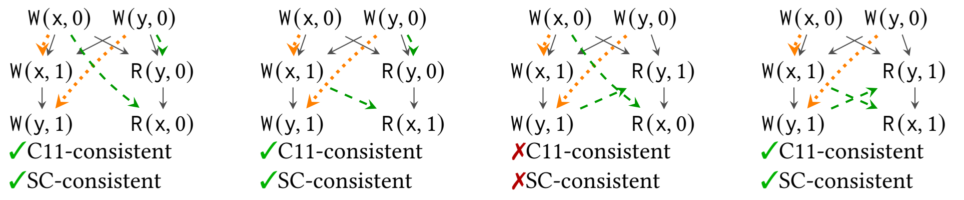
(I) **Store-buffering:**

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



(II) **Message-passing:**

$$\begin{array}{l||l} \mathtt{x := 1} & \mathtt{a := y} \\ \mathtt{y := 1} & \mathtt{b := x} \end{array} \qquad \text{(MP)}$$



Below each test we illustrate candidate execution graphs, assuming all locations are initialized to 0. The nodes represent shared-memory accesses, and the edges represent relations between the accesses: "program order" (*po*), denoted by gray edges, relates accesses according to their order in the program; "reads-from" (*rf*), denoted by green edges, relates every read event to the write event from which it obtains its value; and "modification order" (*mo*), denoted by orange edges, totally orders the writes to each memory location, determining the order in which these writes can be observed. C11-consistent graphs correspond to behaviors allowed by C11; SC-consistent graphs are those allowed by SC.
Observe that SB is *not* robust, as it has a C11-consistent execution that is SC-inconsistent, whereas MP is robust: all its C11-consistent executions are SC-consistent. If we weaken the memory ordering of any of the accesses to y in MP, then it would not be robust, as the third graph would become C11-consistent.

Fig. 1. Examples of consistency and robustness.

and events of the same thread are totally ordered by their serial numbers. For a set E' of events, we write $G.E'$ for $G.\mathsf{E} \cap E'$ (e.g., $G.\mathsf{W} = G.\mathsf{E} \cap \mathsf{W}$).

Derived Relations. We use two derived relations in execution graphs:[2]

$$G.\mathtt{hb} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf})^{+} \quad (\textit{happens-before}) \qquad G.\mathtt{fr} \triangleq G.\mathtt{rf}^{-1}\,;G.\mathtt{mo} \quad (\textit{from-read})$$

Happens-before represents causality among events. The "from-read" relation holds between a read event r and a write event w when r reads from a write w' that is before w in the modification order.

From Programs to Execution Graphs. A concurrent program P induces a labeled transition system (LTS), where each transition is labeled by a pair $\langle \tau, l\rangle \in \mathsf{Tid} \times \mathsf{Lab}$. This LTS is independent of memory consistency and includes a non-deterministic choice among all possible values in Val for every read instruction in P. Its runs induce *candidate execution graphs* of P, where the events of

[2]Given a relation R, $R^?$ and R^+ denote its reflexive and transitive closures. The inverse of R is denoted by R^{-1}, and the (left) composition of two relations R_1, R_2 is denoted by $R_1\,;R_2$.

each thread are determined according to the labels along the run, and the reads-from relations and modification orders are arbitrary. We note that each graph corresponds to a *prefix* of a run of P. The full definitions, which are standard, are omitted (see, e.g., [Lahav and Margalit 2019]).

Consistency. Among all candidate execution graphs of a program, only *consistent* ones correspond to allowed program behaviors. For the fragment discussed here, an execution graph G is *consistent* if the following hold:

- $G.\mathsf{mo}\,;G.\mathsf{hb}$ is irreflexive. (WRITE COHERENCE)
- $G.\mathtt{fr}\,;G.\mathsf{hb}$ is irreflexive. (READ COHERENCE)
- $G.\mathsf{po} \cup G.\mathtt{rf}$ is acyclic. (ACYC-PO-RF)

For instance, the third execution graph of MP in Fig. 1 is inconsistent, as it violates READ COHERENCE. Specifically, there is a $G.\mathsf{hb}$ relation from the event labeled $\mathtt{W}(x,1)$ to the event labeled $\mathtt{R}(x,0)$, while $G.\mathtt{fr}$ between these events points in the opposite direction.

2.2 Robustness

Following previous work, we define robustness using execution graphs. A program P is considered *robust* against a (declarative) memory model M if every M-consistent execution graph generated by P is also SC-consistent. SC-consistency of an execution graph G means that $G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathsf{mo}$ can be linearized into a sequence of events such that every $G.\mathtt{rf}$-edge connects each read r with the last write w to the same location that occurs before r. An equivalent definition from [Alglave et al. 2014] is provided next.

Definition 2.1. An execution graph G is SC-*consistent* if $G.\mathsf{hb}_{\mathrm{SC}}$ is irreflexive, where:

$$G.\mathsf{hb}_{\mathrm{SC}} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathsf{mo} \cup G.\mathtt{fr})^{+} \qquad \text{(SC-}\textit{happens-before}\text{)}$$

For instance, the fourth execution graph of SB in Fig. 1 is SC-inconsistent. It has a cycle following $\mathtt{fr}$, po, $\mathtt{fr}$, and po between the events labelled $\mathtt{R}(\mathtt{x},0)$, $\mathtt{W}(\mathtt{x},1)$, $\mathtt{R}(\mathtt{y},0)$, $\mathtt{W}(\mathtt{y},1)$, and back to $\mathtt{R}(\mathtt{x},0)$.

The SC-happens-before relation, $\mathsf{hb}_{\mathrm{SC}}$, holds between events e_1 and e_2 iff the operation associated with e_1 has to be executed before the one associated with e_2 in every operational SC run that generates G, i.e., in every operational SC run in which threads execute the operations associated with the events in G following their order in $G.\mathsf{po}$; every read r reads its value from the write w for which $\langle w, r\rangle \in G.\mathtt{rf}$; and writes to each location are executed following their order in $G.\mathsf{mo}$.

2.3 Robustness Verification

Lahav and Margalit [2019] studied the problem of static verification of robustness for a given program. The challenge lies in the fact that, although one typically assumes a finite-state programs (as we do here), such programs may still have loops, and thus have infinitely many consistent execution graphs, making it impossible to enumerate them all. Still, Lahav and Margalit [2019] provided a decision procedure for this problem by reducing it to reachability in a *finite*-state machine. Next, we summarize the key insights from their work.

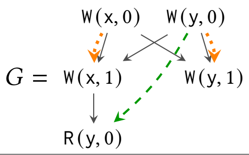
Non-Robustness Witnesses. The first key idea is that robustness can be verified by considering only runs under SC. In essence, when searching for executions that deviate from SC, it suffices to consider "borderline executions": execution graphs of the program that are still SC-consistent, but the next step of the program that will add one more event to the graph may result in an SC-inconsistent graph while maintaining the consistency conditions of the weak memory model. Such executions are called *non-robustness witnesses* of the given program. For the formal definition, we let $G.\mathsf{w}_x^{\mathsf{max}} \triangleq \max_{G.\mathsf{mo}} G.\mathsf{W}_x$ (the most recent write to location x in an execution graph G). Then, a non-robustness witness is a configuration obtained during a run under SC in which some thread τ is about to access (either read from or write to) a location x and the following hold for the (SC-consistent) execution graph G that was generated until this point:

(1) The next access of τ races with $G.\mathsf{w}_x^{\max}$, i.e., thread τ is not $G.\mathsf{hb}$-"aware" of $G.\mathsf{w}_x^{\max}$. Formally, this means that there is no $G.\mathsf{hb}$ from $G.\mathsf{w}_x^{\max}$ to some $e \in G.\mathsf{E}$ with $\mathtt{tid}(e) = \tau$.

(2) The next access of τ cannot have been executed before $G.\mathsf{w}_x^{\max}$, i.e., it must be executed after $G.\mathsf{w}_x^{\max}$ in any SC run that would generate G. Formally, this means that we have $G.\mathsf{hb}_{\mathrm{SC}}$ from $G.\mathsf{w}_x^{\max}$ to some $e \in G.\mathsf{E}$ with $\mathtt{tid}(e) = \tau$.

The first condition ensures that under the C11 memory model thread τ can: (i) read a stale value from x if it tries to read x; and (ii) write a value that is not overwriting the last write to x if it writes to x. The second condition ensures that no execution under SC can generate an execution graph where such access is performed. Lahav and Margalit [2019] show that a program P is not robust *iff* there is some run of P that reaches a non-robustness witness.

Example 2.1. Consider again the SB program from Fig. 1. Under SC, if we execute T_1 first, and then the write instruction of T_2, we reach the following "borderline" execution graph:



With this graph we obtain a non-robustness witness for SB. Indeed, (i) $G.\mathsf{w}_\mathsf{x}^{\max} = \max_{G.\mathsf{mo}} G.\mathsf{W}_\mathsf{x}$ is the write event labeled $\mathtt{W}(\mathtt{x}, 1)$; (ii) the next access of T_2 accesses location $\mathtt{x}$; (iii) there is no $G.\mathsf{hb}$ from $G.\mathsf{w}_\mathsf{x}^{\max}$ to T_2; but (iv) there is $G.\mathsf{hb}_{\mathrm{SC}}$ from $G.\mathsf{w}_\mathsf{x}^{\max}$ to T_2 (via an $G.\mathtt{fr}$-edge from the event labeled $\mathtt{R}(\mathtt{y}, 0)$ to the one labeled $\mathtt{W}(\mathtt{y}, 1)$).

Bounded Instrumentation. The second key observation is that for detecting non-robustness witnesses, it is unnecessary to retain the entire SC-consistent execution graph generated thus far. Instead, maintaining a constant-size summary of the graph is sufficient. Concretely, at each step, we need to know for every thread τ and location x: (i) whether there is $G.\mathsf{hb}$ from $G.\mathsf{w}_x^{\max}$ to thread τ; and (ii) whether there is $G.\mathsf{hb}_{\mathrm{SC}}$ from $G.\mathsf{w}_x^{\max}$ to thread τ. This information can be stored in two "Boolean matrices" (BMs, for short) $\mathrm{T}_{\mathrm{HB}}, \mathrm{T}_{\mathrm{SC}} : \mathsf{Tid} \to 2^{\mathsf{Loc}}$, such that:

$$x \in \mathrm{T}_{\mathrm{HB}}(\tau) \Leftrightarrow \exists e \in G.\mathsf{E}^\tau \cup \mathsf{Init}.\, \langle G.\mathsf{w}_x^{\max}, e\rangle \in G.\mathsf{hb}^?$$
$$x \in \mathrm{T}_{\mathrm{SC}}(\tau) \Leftrightarrow \exists e \in G.\mathsf{E}^\tau \cup \mathsf{Init}.\, \langle G.\mathsf{w}_x^{\max}, e\rangle \in G.\mathsf{hb}_{\mathrm{SC}}^?$$

In turn, three additional BMs, $\mathrm{W}_{\mathrm{HB}}, \mathrm{W}_{\mathrm{SC}}, \mathrm{M}_{\mathrm{SC}} : \mathsf{Loc} \to 2^{\mathsf{Loc}}$, carry sufficient information for faithfully updating T_{HB} and T_{SC} in every execution step:

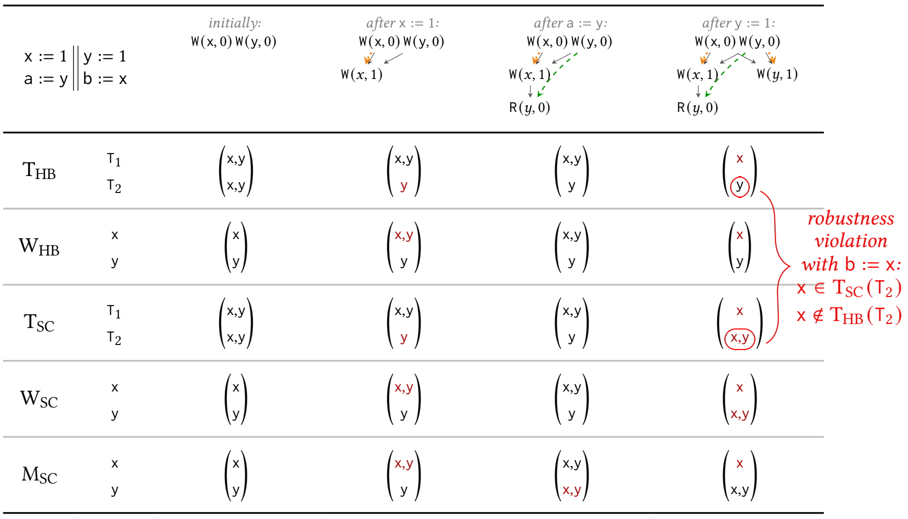
$$x \in \mathrm{W}_{\mathrm{HB}}(y) \Leftrightarrow \langle G.\mathsf{w}_x^{\max}, G.\mathsf{w}_y^{\max}\rangle \in G.\mathsf{hb}^? \qquad x \in \mathrm{W}_{\mathrm{SC}}(y) \Leftrightarrow \langle G.\mathsf{w}_x^{\max}, G.\mathsf{w}_y^{\max}\rangle \in G.\mathsf{hb}_{\mathrm{SC}}^?$$
$$x \in \mathrm{M}_{\mathrm{SC}}(y) \Leftrightarrow \exists e \in G.\mathsf{E}_y.\, \langle G.\mathsf{w}_x^{\max}, e\rangle \in G.\mathsf{hb}_{\mathrm{SC}}^?$$

Initially, $\mathrm{T}_{\mathrm{HB}} = \mathrm{T}_{\mathrm{SC}} = \lambda\tau.\,\mathsf{Loc}$ and $\mathrm{W}_{\mathrm{HB}} = \mathrm{W}_{\mathrm{SC}} = \mathrm{M}_{\mathrm{SC}} = \lambda x.\,\{x\}$. The maintenance of the instrumentation with each access to a shared variable is given in Fig. 2. The rows correspond to operations performed by the program, and each column describes how the instrumentation is updated. When thread τ writes to location x, the location x is added to $\mathrm{T}_{\mathrm{HB}}(\tau)$, and removed from $\mathrm{T}_{\mathrm{HB}}(\pi)$ for all other threads $\pi \neq \tau$. Indeed, after the write, τ is aware of the most recent write to x but no other thread is aware of it. In addition, we assign $\mathrm{W}_{\mathrm{HB}}(x) := \mathrm{T}_{\mathrm{HB}}(\tau)$ as the thread releases its view via the write to x, and for all other locations $y \neq x$, the location x is removed from $\mathrm{W}_{\mathrm{HB}}(y)$. In turn, when τ reads from location x, it acquires the last view released to x, so we add $\mathrm{W}_{\mathrm{HB}}(x)$ to $\mathrm{T}_{\mathrm{HB}}(\tau)$. The SC-BMs update rules are more involved, but are generally similar. In particular, in order to track the $\mathtt{fr}$ order (which is a part of $\mathsf{hb}_{\mathrm{SC}}$), read events propagate the thread view T_{SC} to M_{SC}, which is acquired by later writes to the same location.

	$\mathrm{T_{HB}}(\tau) :=$	for $\pi \neq \tau$ $\mathrm{T_{HB}}(\pi) :=$	$\mathrm{W_{HB}}(x) :=$	for $y \neq x$ $\mathrm{W_{HB}}(y) :=$
Read	$\mathrm{T_{HB}}(\tau) \cup \mathrm{W_{HB}}(x)$	$\mathrm{T_{HB}}(\pi)$	$\mathrm{W_{HB}}(x)$	$\mathrm{W_{HB}}(y)$
Write	$\mathrm{T_{HB}}(\tau) \cup \{x\}$	$\mathrm{T_{HB}}(\pi) \setminus \{x\}$	$\mathrm{T_{HB}}(\tau) \cup \{x\}$	$\mathrm{W_{HB}}(y) \setminus \{x\}$

	$\mathrm{T_{SC}}(\tau) :=$	for $\pi \neq \tau$ $\mathrm{T_{SC}}(\pi) :=$	$\mathrm{W_{SC}}(x) :=$	for $y \neq x$ $\mathrm{W_{SC}}(y) :=$	$\mathrm{M_{SC}}(x) :=$	for $y \neq x$ $\mathrm{M_{SC}}(y) :=$
Read	$\mathrm{T_{SC}}(\tau) \cup \mathrm{W_{SC}}(x)$	$\mathrm{T_{SC}}(\pi)$	$\mathrm{W_{SC}}(x)$	$\mathrm{W_{SC}}(y)$	$\mathrm{M_{SC}}(x) \cup \mathrm{T_{SC}}(\tau)$	$\mathrm{M_{SC}}(y)$
Write	$\mathrm{T_{SC}}(\tau) \cup \mathrm{M_{SC}}(x)$	$\mathrm{T_{SC}}(\pi) \setminus \{x\}$	$\mathrm{T_{SC}}(\tau) \cup \mathrm{M_{SC}}(x)$	$\mathrm{W_{SC}}(y) \setminus \{x\}$	$\mathrm{M_{SC}}(x) \cup \mathrm{T_{SC}}(\tau)$	$\mathrm{M_{SC}}(y) \setminus \{x\}$

Fig. 2. Maintaining BMs when thread τ accesses location x



x := 1 ‖ y := 1
a := y ‖ b := x
		initially:
		W(x, 0) W(y, 0)
			after x := 1:
			W(x, 0) W(y, 0)
			W(x, 1)
				after a := y:
				W(x, 0) W(y, 0)
				W(x, 1)
				R(y, 0)
					after y := 1:
					W(x, 0) W(y, 0)
					W(x, 1) W(y, 1)
					R(y, 0)

$\mathrm{T_{HB}}$	T_1 T_2	$\begin{pmatrix} x,y \\ x,y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x \\ \bigcirc{y} \end{pmatrix}$
$\mathrm{W_{HB}}$	x y	$\begin{pmatrix} x \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x \\ y \end{pmatrix}$
$\mathrm{T_{SC}}$	T_1 T_2	$\begin{pmatrix} x,y \\ x,y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x \\ \bigcirc{x,y} \end{pmatrix}$
$\mathrm{W_{SC}}$	x y	$\begin{pmatrix} x \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x \\ x,y \end{pmatrix}$
$\mathrm{M_{SC}}$	x y	$\begin{pmatrix} x \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ y \end{pmatrix}$	$\begin{pmatrix} x,y \\ x,y \end{pmatrix}$	$\begin{pmatrix} x \\ x,y \end{pmatrix}$

robustness violation with b := x:
x $\in \mathrm{T_{SC}}(\mathsf{T}_2)$
x $\notin \mathrm{T_{HB}}(\mathsf{T}_2)$

Fig. 3. BM-instrumentation maintenance for a run of SB

Robustness Verification. All in all, [Lahav and Margalit 2019] established the following result:

Theorem 2.2. *A program P is robust iff running P under* SC *with the instrumentation given in Fig. 2 never reaches a state in which thread τ is about to access some location $x \in \mathrm{T_{SC}}(\tau) \setminus \mathrm{T_{HB}}(\tau)$.*

In this theorem, running under SC means using the standard interleaving-based operational semantics with the memory state represented as a mapping from variables to values.

Example 2.2. Consider again SB from Fig. 1, and its run described in Ex. 2.1. The BM instrumentation after each instruction is depicted in Fig. 3. When T_1 executes x := 1, we remove x from views associated with T_2. Then, T_1 reads 0 from y and we add x to $\mathrm{M_{SC}}(\mathsf{y})$ since this read of y is $\mathrm{hb_{SC}}$-after the last write to x. When T_2 then executes y := 1, it gains the 'knowledge' from y, and we add x to $\mathrm{T_{SC}}(\mathsf{T}_2)$. Finally, when T_2 tries to read from x we flag a robustness violation as: $\mathsf{x} \in \mathrm{T_{SC}}(\mathsf{T}_2) \setminus \mathrm{T_{HB}}(\mathsf{T}_2)$.

3 Location-Clock-Based Robustness Analysis

The instrumentation in Fig. 2, previously utilized for static robustness verification, can also be directly used in a dynamic analysis. However, this approach presents significant limitations, as detailed in §3.1. Instead, we introduce a novel instrumentation for robustness in §3.2, which is better suited for dynamic analysis.

3.1 Drawbacks of BMs for Dynamic Analysis

The BM instrumentation was specifically developed for static verification, and, as is, it is not suitable for a dynamic analysis. This stems from two reasons.

Blocking Instrumentation. For static verification, it is essential that the instrumentation remains constant in size (assuming a fixed number of threads and memory locations). Constant-size instrumentation, like the BM instrumentation, is necessary for reducing the robustness verification problem to a reachability problem in a finite-state machine. In turn, for a dynamic approach, a constant-size instrumentation is unnecessary and different requirements come into play. A primary consideration in this setting is the extent to which concurrency is compromised in the instrumented program compared to the uninstrumented one. In this regard, BMs pose significant drawbacks: any write operation necessitates updating the BMs across *all threads and locations* (see Fig. 2), as they all lose "awareness" of the last write. This update must occur atomically, thereby blocking concurrent threads from writing simultaneously. Consequently, maintaining the BMs effectively serializes all memory writes, severely limiting parallelism in the original program.

Non-Predictive Instrumentation. For static verification, there is no need to *predict* the existence of a non-robustness witness from an execution graph that itself is not a witness, as we anyway exhaustively account for all possible executions. However, dynamic analysis, by nature, observes specific executions, and the inability to predict a non-robustness witness is a significant drawback. Without predictive capabilities, the analysis might miss non-robustness witnesses that rarely appear in the observed executions. The next example demonstrates this point.

Example 3.1. Consider the following variant of SB:

```
x := 1 || y := 1
a := y || b := x
x := 2 || y := 2
```

This program is not robust: as with SB, it is possible under weak memory for both threads to read the initial value of 0, an outcome not permitted under SC.

The BM-based algorithm is complete and, indeed, it can detect a non-robustness witness in certain runs. For instance, in a run where T_1 executes its first two instructions and then T_2 follows, just before $\mathsf{b} := \mathsf{x}$, we observe that $\mathsf{x} \in \mathrm{T}_{\mathrm{SC}}(\mathsf{T}_2) \setminus \mathrm{T}_{\mathrm{HB}}(\mathsf{T}_2)$ and flag a robustness violation.

However, consider a run in which thread T_1 terminates before T_2 begins. In this sequence, the last write to x when T_2 executes is the event from $\mathsf{x} := 2$. Since there is no $\mathsf{hb}_{\mathrm{SC}}$ path connecting this write event to thread T_2, we find $\mathsf{x} \notin \mathrm{T}_{\mathrm{SC}}(\mathsf{T}_2)$, meaning that BM does not detect a violation in this run. This reveals a limitation of Thm. 2.2 for dynamic verification: even when a prefix of the execution graph witnesses a robustness violation, the current run itself does not flag this issue.

In fact, running a BM-based dynamic analysis on this program over 1,000 executions did not detect the violation. This is because the violation is captured in the BM-based analysis only with a context switch after one thread's first two instructions. Given the short program length, such a context switch is rare without explicitly adjusting the OS scheduling.

3.2 Location Clocks to the Rescue

To overcome the limitations of BMs, we introduce *location clocks*, a variant of vector clocks, and show how to adapt the instrumentation to use location clocks. Location clocks (LCs) have two primary advantages over over BMs. First, LCs are more efficient when maintaining instrumentation, as a thread accessing a location only needs to update the location clocks associated with that

	$\mathbb{T}_{\text{HB}}(\tau) :=$	for $\pi \neq \tau$ $\mathbb{T}_{\text{HB}}(\pi) :=$	$\mathbb{W}_{\text{HB}}(x) :=$	for $y \neq x$ $\mathbb{W}_{\text{HB}}(y) :=$
Read	$\mathbb{T}_{\text{HB}}(\tau) \sqcup \mathbb{W}_{\text{HB}}(x)$	$\mathbb{T}_{\text{HB}}(\pi)$	$\mathbb{W}_{\text{HB}}(x)$	$\mathbb{W}_{\text{HB}}(y)$
Write	$\mathbb{T}_{\text{HB}}(\tau) \sqcup (x@t)$	$\mathbb{T}_{\text{HB}}(\pi)$	$\mathbb{T}_{\text{HB}}(\tau) \sqcup (x@t)$	$\mathbb{W}_{\text{HB}}(y)$

	$\mathbb{T}_{\text{SC}}(\tau) :=$	for $\pi \neq \tau$ $\mathbb{T}_{\text{SC}}(\pi) :=$	$\mathbb{W}_{\text{SC}}(x) :=$	for $y \neq x$ $\mathbb{W}_{\text{SC}}(y) :=$	$\mathbb{M}_{\text{SC}}(x) :=$	for $y \neq x$ $\mathbb{M}_{\text{SC}}(y) :=$
Read	$\mathbb{T}_{\text{SC}}(\tau) \sqcup \mathbb{W}_{\text{SC}}(x)$	$\mathbb{T}_{\text{SC}}(\pi)$	$\mathbb{W}_{\text{SC}}(x)$	$\mathbb{W}_{\text{SC}}(y)$	$\mathbb{M}_{\text{SC}}(x) \sqcup \mathbb{T}_{\text{SC}}(\tau)$	$\mathbb{M}_{\text{SC}}(y)$
Write	$\mathbb{T}_{\text{SC}}(\tau) \sqcup \mathbb{M}_{\text{SC}}(x) \sqcup (x@t)$	$\mathbb{T}_{\text{SC}}(\pi)$	$\mathbb{T}_{\text{SC}}(\tau) \sqcup \mathbb{M}_{\text{SC}}(x) \sqcup (x@t)$	$\mathbb{W}_{\text{SC}}(y)$	$\mathbb{M}_{\text{SC}}(x) \sqcup \mathbb{T}_{\text{SC}}(\tau) \sqcup (x@t)$	$\mathbb{M}_{\text{SC}}(y)$

Fig. 4. Maintaining LCs when thread τ accesses location x, where $t = \mathbb{W}_{\text{HB}}(x)(x) + 1 = \mathbb{W}_{\text{SC}}(x)(x) + 1$

specific thread and location. This is critical for dynamic analysis, as it enables locking only the instrumentation relevant to the accessed location, and such fine-grained locking allows for significantly higher concurrency during the (instrumented) program execution. Second, LCs support the *prediction* of non-robustness witnesses, effectively addressing the challenge illustrated in Ex. 3.1. We now proceed with the formal definitions.

Location Clocks. Let Time denote the set of natural numbers. We refer to the elements of Time as *timestamps* and use t to range over them. A *location epoch* is a pair $x@t$ where $x \in$ Loc and $t \in$ Time, and a *location clock* (or LC) $\mathbb{L}$ is a set of location epochs such that each location x appears in at most one location epoch in $\mathbb{L}$. We identify a location epoch $x@t$ with a singleton LC $\{x@t\}$. An LC $\mathbb{L}$ essentially represents a function from Loc to Time, assigning the (unique) timestamp t such that $x@t \in \mathbb{L}$ for every $x \in$ Loc that appears in $\mathbb{L}$, and 0 for every other $x \in$ Loc. We often identify LCs with the functions they represent, writing, e.g., $\mathbb{L}(x)$ for the timestamp $\mathbb{L}$ assigns to x. The empty LC, representing the function $\lambda x.\, 0$, is denoted by $\mathbb{L}_{\text{Init}}$, and we let $\mathbb{L}_1 \sqcup \mathbb{L}_2 \triangleq \lambda x.\, \max\{\mathbb{L}_1(x), \mathbb{L}_2(x)\}$.

Location-Clock Instrumentation for Robustness. LCs can replace BMs for robustness verification. Intuitively, each write to a location x obtains a monotonically increasing timestamp, and we keep track of the last timestamp for each location that each thread "knows about" in the hb and hb_{SC} relations. Formally, given an execution graph G, the modification order $G.\text{mo}$ is interpreted as a timestamp mapping $G.\text{ts} : G.\text{W} \to \text{Time}$ defined by $G.\text{ts}(w) \triangleq |\{w' \in G.\text{W} \mid \langle w', w \rangle \in G.\text{mo}\}|$, i.e., the number of write events that are mo-before w. The instrumentation consists of mappings: $\mathbb{T}_{\text{HB}}$ and $\mathbb{T}_{\text{SC}}$ that assign an LC to every thread; and $\mathbb{W}_{\text{HB}}$, $\mathbb{W}_{\text{SC}}$, and $\mathbb{M}_{\text{SC}}$ that assign an LC to every location. The meaning of these LCs for a given execution graph G is summarized as follows:

$$\mathbb{T}_{\text{HB}}(\tau)(x) = \max\{G.\text{ts}(w) \mid w \in \text{W}_x \land \exists e \in G.\text{E}^{\tau}.\, \langle w, e \rangle \in G.\text{hb}^{?}\}$$
$$\mathbb{W}_{\text{HB}}(y)(x) = \max\{G.\text{ts}(w) \mid w \in \text{W}_x \land \langle w, G.\text{w}_y^{\max} \rangle \in G.\text{hb}^{?}\}$$
$$\mathbb{T}_{\text{SC}}(\tau)(x) = \max\{G.\text{ts}(w) \mid w \in \text{W}_x \land \exists e \in G.\text{E}^{\tau}.\, \langle w, e \rangle \in G.\text{hb}_{\text{SC}}^{?}\}$$
$$\mathbb{W}_{\text{SC}}(y)(x) = \max\{G.\text{ts}(w) \mid w \in \text{W}_x \land \langle w, G.\text{w}_y^{\max} \rangle \in G.\text{hb}_{\text{SC}}^{?}\}$$
$$\mathbb{M}_{\text{SC}}(y)(x) = \max\{G.\text{ts}(w) \mid w \in \text{W}_x \land \exists e \in G.\text{E}_y.\, \langle w, e \rangle \in G.\text{hb}_{\text{SC}}^{?}\}$$

Above we take $\max \emptyset \triangleq 0$. Note that for every $x \in$ Loc, $\mathbb{W}_{\text{HB}}(x)(x) = \mathbb{W}_{\text{SC}}(x)(x)$, both storing the maximal timestamp among all writes to x (which is $|G.\text{W}_x| - 1$).

Initially, all LCs are initialized to $\mathbb{L}_{\text{Init}}$. The maintenance of the instrumentation with each access to a shared variable is given in Fig. 4. The rows correspond to operations performed by the program, while each column describes how the instrumentation is updated. For example, when thread τ reads from location x, τ incorporates x's view into its own view, thus "learning" whatever x "knew".

When τ writes to location x, the timestamp for x is incremented, and added to $\mathbb{T}_{\text{HB}}(\tau)$. In addition, τ "releases its knowledge", sharing it with the location so that any thread that will read x in the future will be able to absorb it. As a side effect, due to increment of the timestamp, all other locations no longer have the maximal timestamp for x and thus are unaware of the last write to x. The SC transitions follow the same intuitions. In particular, to track $\mathtt{fr}$, read events propagate the thread view $\mathbb{T}_{\text{SC}}$ to $\mathbb{M}_{\text{SC}}$, which is acquired by later writes to the same location.

To test for robustness violation, when a thread accesses x, we compare the maximal timestamp for x the thread "knows about" in hb_{SC} and in hb. If the first is strictly greater, we flag a violation.

Theorem 3.1. *A program P is robust iff running P under* SC *with the instrumentation given in Fig. 4 never reaches a state in which thread τ is about to access location x but $\mathbb{T}_{\text{HB}}(\tau)(x) < \mathbb{T}_{\text{SC}}(\tau)(x)$.*

Proof (sketch). We first show that the LC instrumentation indeed tracks the timestamps as described above. Then, if P is not robust, then by [Lahav and Margalit 2019, Thm. 5.1.], there exists a non-robustness witness for P, which implies that under SC P generates some (SC-consistent) execution graph G with $\langle G.\mathsf{w}_x^{\max}, e\rangle \in G.\mathsf{hb}_{\text{SC}}$ for some $e \in G.\mathsf{E}^\tau$ but $\langle G.\mathsf{w}_x^{\max}, e\rangle \notin G.\mathsf{hb}$ for every $e \in G.\mathsf{E}^\tau$, and the next step of thread τ is an access to location x. In that run, we will have $\mathbb{T}_{\text{HB}}(\tau)(x) < \mathbb{T}_{\text{SC}}(\tau)(x) = G.\mathsf{ts}(G.\mathsf{w}_x^{\max})$.

For the converse, we first note that a run of P under SC that reaches a state in which thread τ is about to access location x but $\mathbb{T}_{\text{HB}}(\tau)(x) < \mathbb{T}_{\text{SC}}(\tau)(x)$ implies the existence of an *extended non-robustness witness*. The latter is defined as a configuration in which some thread τ is about to access a location x and the following hold for the (SC-consistent) generated execution graph G and some $w \in G.\mathsf{W}_x$:

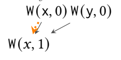
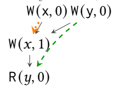
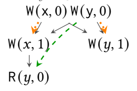
(1) The next access of τ races with w, as well as with every write w' that is $G.\mathsf{mo}$-after w, i.e., $\langle w, e\rangle \notin G.\mathsf{mo}^? \,; G.\mathsf{hb}$ for every $e \in \mathsf{E}^\tau$.
(2) The next access of τ cannot have been executed before w, i.e., $\langle w, e\rangle \in G.\mathsf{hb}_{\text{SC}}$ for some $e \in \mathsf{E}^\tau$.

Now, the existence of an extended non-robustness witness implies a standard non-robustness witness, from which by [Lahav and Margalit 2019, Thm. 5.1.], it follows that P is not robust. Indeed, let w_0 be the $G.\mathsf{mo}$-maximal such write event, and construct an execution graph G' by restricting G to all nodes e' such that $\langle e', e\rangle \in G.\mathsf{hb}_{\text{SC}}$. It is easy to see that there exists another run of P under SC that reaches a state in which thread τ is about to access location x, and the generated graph in this run is G'. Moreover, we have $G'.\mathsf{w}_x^{\max} = w_0$, and so $\langle G'.\mathsf{w}_x^{\max}, e\rangle \in G'.\mathsf{hb}_{\text{SC}}$ for some $e \in G'.\mathsf{E}^\tau$ but $\langle G'.\mathsf{w}_x^{\max}, e\rangle \notin G'.\mathsf{hb}$ for every $e \in G'.\mathsf{E}^\tau$. □

Theorem 3.1 provides the formal basis for the guarantees of the dynamic analysis. The '⇒' direction ensures soundness, meaning no spurious violations. The converse ensures completeness, indicating that for a non-robust program, some schedule exposes the violation.

Example 3.2. Consider again SB from Fig. 1, and its run described in Ex. 2.1. The LC instrumentation after each instruction is depicted in Fig. 5. When $\mathtt{T}_1$ executes $\mathtt{x := 1}$, we increment $\mathtt{x}$ timestamp and update the LCs for $\mathtt{T}_1$ and $\mathtt{x}$. Then, $\mathtt{T}_1$ reads 0 for $\mathtt{y}$ and we update the timestamp for $\mathtt{x}$ in $\mathbb{M}_{\text{SC}}(\mathtt{y})$ since the read of $\mathtt{y}$ is aware of the last write to $\mathtt{x}$. When $\mathtt{T}_2$ then executes $\mathtt{y := 1}$, it learns about the events from $\mathtt{y}$, specifically the last write to $\mathtt{x}$. Finally, when $\mathtt{T}_2$ tries to read from $\mathtt{x}$ we flag a robustness violation as: $\mathbb{T}_{\text{HB}}(\mathtt{T}_2)(\mathtt{x}) < \mathbb{T}_{\text{SC}}(\mathtt{T}_2)(\mathtt{x})$. To see the advantage over BMs compare with Ex. 2.2. For instance, when $\mathtt{T}_2$ executes $\mathtt{y := 1}$, we only update the LCs associated with $\mathtt{T}_2$ and $\mathtt{y}$, whereas in the BM instrumentation we needed to update almost all BMs.

We note that the space required for the LC instrumentation is $O((|\mathsf{Tid}| + |\mathsf{Loc}|) \cdot |\mathsf{Loc}| \cdot \log n)$, where n represents the number of write events in the analyzed run. The $\log n$ factor arises from

x := 1 ‖ y := 1 a := y ‖ b := x		*initially:* W(x, 0) W(y, 0)	*after* x := 1: 	*after* a := y: 	*after* y := 1: 
$\mathbb{T}_{HB}$	T_1	{x@0,y@0}	{x@1,y@0}	{x@1,y@0}	{x@1,y@0}
	T_2	{x@0,y@0}	{x@0,y@0}	{x@0,y@0}	{x@0,y@1}
$\mathbb{W}_{HB}$	x	{x@0,y@0}	{x@1,y@0}	{x@1,y@0}	{x@1,y@0}
	y	{x@0,y@0}	{x@0,y@0}	{x@0,y@0}	{x@0,y@1}
$\mathbb{T}_{SC}$	T_1	{x@0,y@0}	{x@1,y@0}	{x@1,y@0}	{x@1,y@0}
	T_2	{x@0,y@0}	{x@0,y@0}	{x@0,y@0}	{x@1,y@1}
$\mathbb{W}_{SC}$	x	{x@0,y@0}	{x@1,y@0}	{x@1,y@0}	{x@1,y@0}
	y	{x@0,y@0}	{x@0,y@0}	{x@0,y@0}	{x@1,y@1}
$\mathbb{M}_{SC}$	x	{x@0,y@0}	{x@1,y@0}	{x@1,y@0}	{x@1,y@0}
	y	{x@0,y@0}	{x@0,y@0}	{x@1,y@0}	{x@1,y@1}

robustness violation with b := x*:* $\mathbb{T}_{HB}(T_2)(x) < \mathbb{T}_{SC}(T_2)(x)$

Fig. 5. LC-instrumentation maintenance for a run of SB

the need to record timestamps. In practice, however, timestamps are bounded and stored as large integers (64 bits), making the actual memory consumption independent of the length of the run.

Predictive Analysis Using LCs. We highlight the predictive advantage of Thm. 3.1 over Thm. 2.2 from [Lahav and Margalit 2019]. First, revisiting Ex. 3.1, note that if we run T_1 to completion before T_2 begins, we still have that $\mathbb{T}_{HB}(T_2)(x) < \mathbb{T}_{SC}(T_2)(x)$ when T_2 tries to read from x. Indeed, the final write by T_1 to x does not change the LCs associated with T_2. As a result, the LC-based analysis based on Thm. 3.1 detects the violation in *any run* of the program (as confirmed in our experiments).

More generally, LCs are able to detect *extended* non-robustness witnesses, as defined in the proof of Thm. 3.1, which imply the existence of a non-robustness witness in a different run. One can use LCs to detect standard witnesses (without prediction), by a straightforward adaptation of Thm. 2.2 to LCs, based on the following relationship between the LC and BM instrumentations:

$$\mathbb{T}_{HB}(\tau)(x) = \mathbb{W}_{HB}(x)(x) \Leftrightarrow x \in T_{HB}(\tau) \qquad \mathbb{T}_{SC}(\tau)(x) = \mathbb{W}_{SC}(x)(x) \Leftrightarrow x \in T_{SC}(\tau)$$

Thus, to detect standard witnesses we should flag a violation when thread τ is about to access location x and $\mathbb{T}_{HB}(\tau)(x) < \mathbb{W}_{HB}(x)(x) = \mathbb{W}_{SC}(x)(x) = \mathbb{T}_{SC}(\tau)(x)$, which means that τ is aware of the globally maximal write to x in hb_{SC}, yet unaware of the globally maximal write to x in hb. By contrast, the (weaker) condition $\mathbb{T}_{HB}(\tau)(x) < \mathbb{T}_{SC}(\tau)(x)$ in Thm. 3.1 means that the maximal write to x that τ is hb-aware of is mo-before the maximal write to x that τ is hb_{SC}-aware of. The latter condition allows the LC-based analysis to be predictive, which enhances the non-robustness detection rate.

Relation to Race Detection. We discuss the relation to standard dynamic race-detection that uses vector clocks (see [Mansky et al. 2017] for a formal description). We observe that extended non-robustness witnesses are races on atomic accesses, but, of course, not every such race imply non-robustness. Besides identifying a race on location x between an event e_1 and a later executed event e_2 (i.e., two accesses accessing x such that at least one of them is a write, and $\langle e_1, e_2\rangle \notin G.\mathsf{hb}\,;G.\mathsf{po}$), in extended non-robustness witnesses one also requires that: (i) e_1 has to be a write event (so we do not check for so-called "read-write" races); (ii) every write event to x executed after e_1 races with e_2 (i.e., $\langle e_1, e_2\rangle \notin G.\mathsf{mo} \mathbin{;} G.\mathsf{hb} \mathbin{;} G.\mathsf{po}$); and (iii) e_2 must be executed after e_1 in any run under SC (i.e., $\langle e_1, e_2\rangle \in G.\mathsf{hb}_{\mathsf{SC}} \mathbin{;} G.\mathsf{po}$). Tracking such patterns in a dynamic analysis requires non-trivial extensions of standard race detection. Our technique does so by switching from standard vector clocks (functions from Tid to Time) to location clocks (functions from Loc to Time).

4 Key Enhancements to the Basic Approach

We introduce several essential extensions to the basic approach, broadening its applicability to real-world concurrent programs. We discuss support for read-modify-write instructions (§4.1), extension to the full RC20 mode (§4.2), and a technique for masking benign robustness violations caused by busy loops (§4.3). This discussion remains informal, focusing on the main concepts behind each extension. Details integrating all extensions are provided in [Margalit et al. 2025, §A].

4.1 Supporting Read-Modify-Writes

Our discussion so far has focused on reads and writes, omitting atomic read-modify-write (RMW) instructions, such as fetch-and-add (FADD) and compare-and-swap (CAS)[3], which are indispensable in concurrent programs. Our method extends to support RMWs.

To model RMWs in execution graphs, C11 uses RMW events, labeled with $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$, denoting the location x being updated, the value $v_{\mathtt{R}}$ read from x, and the value $v_{\mathtt{W}}$ that replaces $v_{\mathtt{R}}$. RMW events behave both like reads (e.g., require an incoming $\mathtt{rf}$-edge to justify the value being read) and writes (e.g., totally ordered per location with other writes via mo). The distinctiveness of RMWs, beyond simply combining a read and a write, is captured by another constraint in consistent graphs:[4]

- $G.\mathtt{fr} \mathbin{;} G.\mathsf{mo}$ is irreflexive. (ATOMICITY)

This condition requires that the oncoming $G.\mathtt{rf}$-edge to each RMW event e originates from the *immediate* $G.\mathsf{mo}$-predecessor of e.

Due to the ATOMICITY, supporting RMWs in robustness analysis requires specific adjustments. Indeed, without RMWs, if the next access by thread τ is a write to location x, and some write w to x has no $G.\mathsf{mo}^? \mathbin{;} G.\mathsf{hb}^?$ path to thread τ, we can always place the next write as the immediate predecessor of w in $G.\mathsf{mo}$. This would indicate a robustness violation when w has $G.\mathsf{hb}_{\mathsf{SC}}$ to thread τ, and thus no SC-consistent execution allows this placement. However, with RMWs, ATOMICITY forbids the next write from being placed immediately before w in $G.\mathsf{mo}$ when w itself is an RMW.

To handle RMWs, our approach extends the LC instrumentation by incorporating an additional timestamp at each location, tracking the cumulative count of *non-RMW writes* performed for the location. Each thread and location maintains LCs containing these timestamps as well, reflecting their awareness of this timestamp in hb and $\mathsf{hb}_{\mathsf{SC}}$. Upon each write (or RMW), we use these LCs to detect potential robustness violations (whereas upon reads we still use the standard LCs). Thus, a violation is flagged only if a write can indeed be added to the current graph in a way that preserves weak memory consistency, including ATOMICITY, yet disrupts SC-consistency.

EXAMPLE 4.1. Consider the tests and corresponding candidate execution graphs in Fig. 6. Program (1) is not robust: G_1 is C11-consistent but SC-inconsistent. However, Program (2) is robust. In particular, due to ATOMICITY, G_2 is C11-inconsistent, and thus G_2 does not serve as a counterexample to robustness. Note, however, that if we replace the $\mathtt{a} := \mathbf{FADD}(\mathtt{x}, 1)$ instruction in $\mathtt{T}_1$ with a normal write, Program (2) also becomes non-robust. In this modified case, G_2 with $\mathtt{RMW}(\mathtt{x}, 0, 1)$ replaced by

[3]The extension discussed in this section applies to *weak* CAS, which may fail even when it reads the expected value. To support *strong* CAS, this extension must be combined with the value tracking discussed in §4.3. Our full development, along with the accompanying tool, supports both types of CAS.

[4]With RMW events, the definition of $G.\mathtt{fr}$ has to exclude the identity relation: $G.\mathtt{fr} \triangleq (G.\mathtt{rf}^{-1} \mathbin{;} G.\mathsf{mo}) \setminus \{\langle e, e\rangle \mid e \in \mathsf{E}\}$.

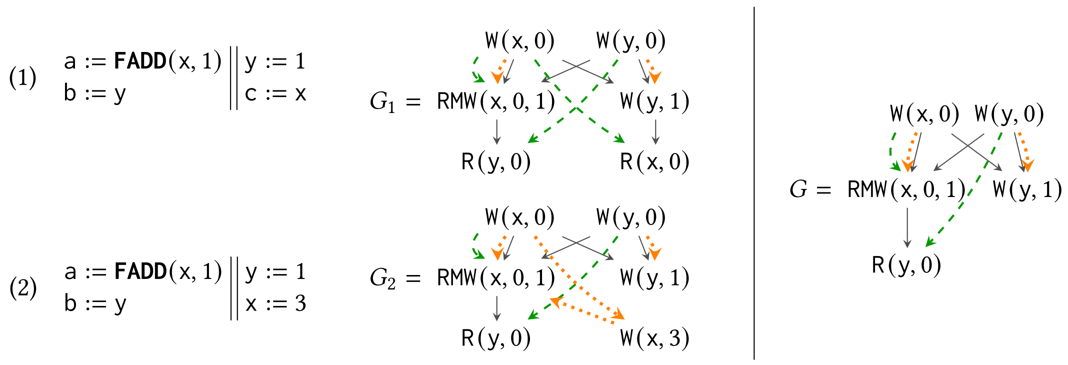


Fig. 6. Examples of programs with RMWs, see Ex. 4.1

the label of the new write (and the rf-incoming edge toward this node removed) would become a counterexample to robustness.

The graph G on the right serves as a witness of non-robustness for Program (1). If we execute T_1 first and then T_2, we reach a scenario where T_1 has the timestamp of the RMW in its $\mathbb{T}_{\text{SC}}$ LC but not in its $\mathbb{T}_{\text{HB}}$ LC. Therefore, to maintain soundness and avoid reporting a spurious violation for Program (2), we adjust the condition being checked: when T_2 is about to perform a write, we compare the RMW LCs $\mathbb{T}_{\text{SC}}^{\text{RMW}}$ to $\mathbb{T}_{\text{HB}}^{\text{RMW}}$, rather than $\mathbb{T}_{\text{SC}}$ to $\mathbb{T}_{\text{HB}}$. The modified LCs do not count RMWs in their timestamps and, for the run discussed, would carry only the initial write in both clocks.

4.2 Supporting RC20

Our approach extends to a substantial fragment of the C11 model that includes, in addition to the release/acquire accesses discussed so far, non-atomic accesses, atomic relaxed accesses, and release/acquire fences. Their semantics follows RC20 as defined in [Margalit and Lahav 2021]. In particular, the "out-of-thin-air" problem is conservatively avoided by disallowing $G.\mathsf{po} \cup G.\mathsf{rf}$ cycles like in RC11 [Lahav et al. 2017]. SC-fences are modeled using release/acquire fences and RMWs to a dedicated, otherwise unused location. SC accesses are not included in this model.

The inclusion of relaxed accesses and release/acquire fences complicates synchronization. Instead of incorporating rf directly into hb, RC20 employs a derived relation, sw (*synchronized with*), which includes selected rf edges along with other synchronization patterns via release/acquire fences and "release sequence" formed by rf edges linked through intermediary RMWs.

To support this extension, we refine the tracking of the hb relation. Rather than the single $\mathbb{T}_{\text{HB}}$ LC, we now maintain three LCs—$\mathbb{T}_{\mathsf{r}}$, $\mathbb{T}_{\mathsf{c}}$, and $\mathbb{T}_{\mathsf{a}}$—that capture the "release", "current", and "acquire" views of each thread, with similar adaptations for the RMW LCs discussed in §4.1. (Notably, the SC LCs that track hb_{SC} remain unchanged.) This approach is inspired by the operational modeling of hb in the Promising Semantics [Kang et al. 2017].

Non-atomics also require attention. Since data races on non-atomics are considered program errors, and since we assume the robustness verification is always complemented by dynamic race detection, the robustness analysis simply ignores non-atomic accesses. But, to faithfully detect data races on non-atomics, we do need to properly track RC20's happens-before relation, hb, with the different synchronization patterns mentioned above. Indeed, a data race on non-atomics means that the conflicting accesses are not related by hb, and while robustness of the atomic accesses in the program allows us to ignore weak memory behaviors for atomics, the whether or not two conflicting non-atomic accesses are synchronized by hb does depend on the way atomics and fences are used (even in SC-consistent graphs). For that matter, we devised a race detection algorithm (given in [Margalit et al. 2025, §B]), based on standard vector clocks, which, in particular, fully supports release/acquire fences. We note that robustness of atomics is essential for the completeness of this algorithm; otherwise, it is affected by the "observer effect" (see Ex. 1.1).

4.3 Avoiding Benign Robustness Violations

In some situations, achieving SC is overly costly, and certain robustness violations can be tolerated. A common example involves a thread in a busy-wait loop, monitoring a specific value—such as a flag set by another thread—before proceeding. Under SC, the loop would terminate when the required value becomes visible in memory. Under weakly consistent memory, the thread may observe a stale value and continue to spin in the loop until the update propagates. A similar scenario is a CAS loop, where a thread repeatedly attempts to atomically update a variable until it succeeds. In weakly consistent models, the CAS may fail and retry even when it would succeed under SC. Such robustness violations are typically considered benign as they do not compromise safety.

To support these benign robustness violations, which were also handled by previous static analysis [Lahav and Margalit 2019], we require additional annotations in the input program. (Static analysis could automate this translation, but it is beyond our current scope.) Busy-wait loops should be explicitly marked. We use two language constructs for this purpose: a **wait**$(x = v)$ instruction, which blocks execution until it can read v from x, and a **BCAS**(x, v_R, v_W) instruction, which blocks until a CAS operation on x succeeds in changing v_R to v_W. These instructions are designed as blocking, ensuring that execution graphs generated from programs using these constructs do not include events for failed attempts. This approach allows us to retain the robustness definition as it is: every RC20-consistent execution graph of the program must also be SC-consistent.

From the annotated code, we compile a finite set of "critical location-value pairs", $\mathsf{C} \subseteq \mathsf{Loc} \times \mathsf{Val}$, and track additional information associated for each pair. Specifically, for each $\langle x, v\rangle \in \mathsf{C}$ and thread $\tau \in \mathsf{Tid}$, we maintain a timestamp $\mathbb{T}_{\mathtt{val}}(\tau)(x, v)$ that records the largest timestamp of a write w_x^v of v to x that occurs mo-before a write w that has $\mathsf{hb}_{\mathsf{SC}}$ to thread τ. When thread τ is about to execute **wait**$(x = v)$, we compare $\mathbb{T}_{\mathrm{HB}}(\tau)(x)$ to $\mathbb{T}_{\mathtt{val}}(\tau)(x, v)$, and flag a violation if $\mathbb{T}_{\mathrm{HB}}(\tau)(x) \leq \mathbb{T}_{\mathtt{val}}(\tau)(x, v)$. This check ensures that not only $\mathbb{T}_{\mathrm{HB}}(\tau)(x) < \mathbb{T}_{\mathrm{SC}}(\tau)(x)$—permitting the thread to read from a stale write under RC20 as before—but also that the stale write indeed holds the required value v. Additionally, we maintain timestamps $\mathbb{W}_{\mathtt{val}}(y)(x, v)$ and $\mathbb{M}_{\mathtt{val}}(y)(x, v)$ to be able to update $\mathbb{T}_{\mathtt{val}}(\tau)(x, v)$ consistently.

Example 4.2. Consider the following BAR program implementing a simple barrier:

```
x := 1            ||  y := 1
while y ≠ 1 do    ||  while x ≠ 1 do        (BAR)
   skip           ||     skip
```

$$G = \begin{tikzcd} \mathtt{W}(x,0) \arrow[d] \arrow[d, dotted] \arrow[dr] & \mathtt{W}(y,0) \arrow[dl] \arrow[d, dotted] \arrow[d] \arrow[ddl, dashed] \\ \mathtt{W}(x,1) \arrow[d] & \mathtt{W}(y,1) \\ \mathtt{R}(y,0) & \end{tikzcd}$$

Program BAR is not robust. Under weak memory semantics, both while loops might repeatedly read 0 for a while, whereas under SC, at least one of them must read 1 in its very first iteration. However, this difference does not impact the barrier's safety, as each thread can only proceed after the other has raised its flag. The execution graph G on the right serves as a non-robustness witness for BAR. In a run where T_1 executes a write and a read before T_2 starts, a robustness violation is detected just before T_2 reads x, as we observe: $0 = \mathbb{T}_{\mathrm{HB}}(\mathsf{T}_2)(\mathtt{x}) < \mathbb{T}_{\mathrm{SC}}(\mathsf{T}_2)(\mathtt{x}) = 1$.

To mask this robustness violation, we expect the user to provide Program $\mathrm{BARW}_{1,1}$ below, which explicitly captures the intended behavior of the busy loops.

```
x := 1        || y := 1            x := 1        || y := 1            x := 1        || y := 1
wait(y = 1)   || wait(x = 1)       wait(y = 0)   || wait(x = 2)       wait(y = 0)   || wait(x = 0)
```

$(\mathrm{BARW}_{1,1})$ $(\mathrm{BARW}_{0,2})$ $(\mathrm{BARW}_{0,0})$

The graph G above is not a non-robustness witness for $\mathrm{BARW}_{1,1}$, as it is not even a candidate execution graph of this program. However, $\mathrm{BARW}_{1,1}$ does not fully illustrate the issue. For example, G is still an execution graph of the *robust* program $\mathrm{BARW}_{0,2}$ above. In $\mathrm{BARW}_{0,2}$, under the described run, we observe: $0 = \mathbb{T}_{\mathrm{HB}}(\mathsf{T}_2)(\mathsf{x}) < \mathbb{T}_{\mathrm{SC}}(\mathsf{T}_2)(\mathsf{x}) = 1$. The specialized tracking of the pair $\langle \mathsf{x}, 2\rangle$ successfully prevents the reporting of such spurious violations: here, we have $\mathbb{T}_{\mathsf{val}}(\mathsf{T}_2)(\mathsf{x}, 2) = -1$ (the initial timestamp we use when no write of 2 to x exists), and $\mathbb{T}_{\mathrm{HB}}(\mathsf{T}_2)(\mathsf{x}) > \mathbb{T}_{\mathsf{val}}(\mathsf{T}_2)(\mathsf{x}, 2)$.

We also note that a naive approach that checks for robustness violations only when **wait** can succeed is flawed. To illustrate, consider the program $\mathrm{BARW}_{0,0}$ above. It is not robust because only weak memory semantics allow *both* **wait** to proceed. Since the dynamic robustness analysis runs the instrumented program under SC, we would never encounter a scenario where both **wait** instructions succeed, thus this naive solution would miss the violation entirely.

5 Implementation

We have implemented our approach in an open-source tool called RSan (Robustness Sanitizer). In this section, we discuss some design choices and optimizations in RSan's implementation.

A dynamic analysis is performed either online ("on-the-fly"), by instrumenting the program for monitoring its executions, or offline ("post-mortem") by first obtaining a program trace, and then analyzing it. The online approach is followed by tools like TSan, while the offline approach is typically followed by experimental predictive race detectors [Kini et al. 2017; Mathur et al. 2020; Pavlogiannis 2020]. We opted for an online analysis in order to quickly and accurately provide users with debugging information (e.g., source-code-level names, stack trace). Although it is possible to store all relevant debugging information for post-mortem analysis, we found that doing so makes parsing and analyzing the trace much slower.

We implemented RSan on top of the TSan framework [ThreadSanitizer 2020]. TSan is a dynamic race detector that ships by default with clang, and can handle C/C++ code. Building on top of TSan's infrastructure allows us to scale to large, real-world programs that use various features of C/C++, including, e.g., thread local storage, pointers, and dynamic thread creation.

To be able to dynamically monitor a program, TSan (and, by extension, RSan) uses a non-invasive instrumentation, meaning that user programs are compiled with a special flag, and there is no need to include custom headers or link against heavyweight libraries. Upon compilation of a program to LLVM-IR, a compiler pass transforms all memory accesses to special function calls, which are used to maintain the required instrumentation. RSan only tracks atomic accesses and fences. For value-tracking (see §4.3), we require users to call RSan-specific primitives corresponding to the **wait**, **BCAS**, and strong **CAS** instructions. A future static analysis could automate the insertion of these calls in many cases.

RSan extends the existing infrastructure of TSan to maintain the LCs described in §3 and §4 (see [Margalit et al. 2025, §A] for the full algorithm). The LC maintenance also ensures SC semantics, since every two accesses to the same location are synchronized via locks. This enables us to formally apply Thm. 3.1. To detect data races on non-atomics, we also modify TSan to execute our race-detection algorithm discussed in §4.2 (given in [Margalit et al. 2025, §B]).

To obtain a scalable implementation, our optimization efforts focused on efficiently implementing LC operations, as real-world programs comprise thousands of locations, and LCs (indexed by locations) are more difficult to handle than vector clocks (indexed by thread identifiers). We implement LCs as sorted arrays of pairs. When looking for a specific address in an LC, we perform a brute-force search if the LC size is small (this yields better cache locality and branch prediction), and fall back to a binary search for larger LCs. To efficiently implement LC merge, we avoid allocating a new LC for the result. Instead, we first update the common addresses of the two LCs in the LHS using a linear pass, we then add the extra RHS addresses to the LHS, and finally re-sort the array. We leave for future work further optimizations in this aspect.

Finally, to enhance user friendliness, we ensured RSan error reports can be used to establish robustness. Whenever a robustness violation is found, RSan reports the offending events to the user. Concretely, an RSan report includes the write event whose timestamp does not satisfy the required condition for robustness, and the concurrent memory access that triggers this check. To assist locating the source of the violation, RSan also provides the corresponding stack traces and source-level instructions.

6 Evaluation

In RSan's evaluation, we aim to demonstrate the following points:

- §6.1 RSan finds almost all robustness violations found by exhaustive (static) techniques, but faster and using less memory;
- §6.3 RSan incurs an acceptable overhead over TSan; and
- §6.2 RSan can detect and repair robustness violations in real-world codebases, often comprising thousands of lines of code, much beyond the state-of-the-art in robustness checking.

To do so, we ran RSan on a representative set of both synthetic and real-world benchmarks. As our evaluation demonstrates, RSan is an excellent alternative to instrumentation-heavy or exhaustive techniques, as (a) it consumes much less memory and scales to many more threads than exhaustive techniques, (b) it detects robustness violations in large codebases within seconds, and (c) requires minimal expertise to be run and repair such violations.

Experimental Setup. All tests were done on a Intel®Xeon® Gold 6132 CPU @2.60GHz GNU/Linux machine with 187GB of RAM. We used a timeout of 60 minutes, and a memory limit of 2GB. Memory consumption is reported in MB and time in seconds. The values are averaged across 10 runs.

6.1 Comparison with Exhaustive Techniques

We begin by comparing RSan with Rocker, a static robustness verifier from [Margalit and Lahav 2021] following the approach described in §2.3. Rocker accepts as input programs written in a toy language called TPL, which are then instrumented, translated to Promela, and verified by the SPIN model checker [Holzmann 1997]. Crucially, the input for Rocker has to be a finite-state program, so exhaustive search converges.

For this part of our evaluation, we restrict ourselves to synthetic benchmarks since (a) translating C/C++ to TPL is arduous, and (b) as our results below show, Rocker cannot handle larger programs anyway. We use synthetic versions of well-known synchronization algorithms obtained from previous work [Margalit and Lahav 2021], including Dekker's mutual exclusion algorithm [Dijkstra 1965] and Peterson's algorithm [Peterson 1981].

We have compared RSan and Rocker on both robust and non-robust benchmarks. First, let us consider the non-robust benchmarks (Fig. 7). For this comparison, we report the average time/memory that RSan and Rocker need to detect a robustness violation, respectively. As can be seen, RSan is able to find robustness violations much more quickly and with much less memory than Rocker. Indeed, while Rocker often needs minutes and gigabytes of memory the report a robustness violation, RSan typically reports violations within seconds and with constant memory consumption. The number of runs required to find a robustness violation in these tests ranges between a single run (`dekker` and `peterson`) to 30 (`eventCounters` for 8 threads).

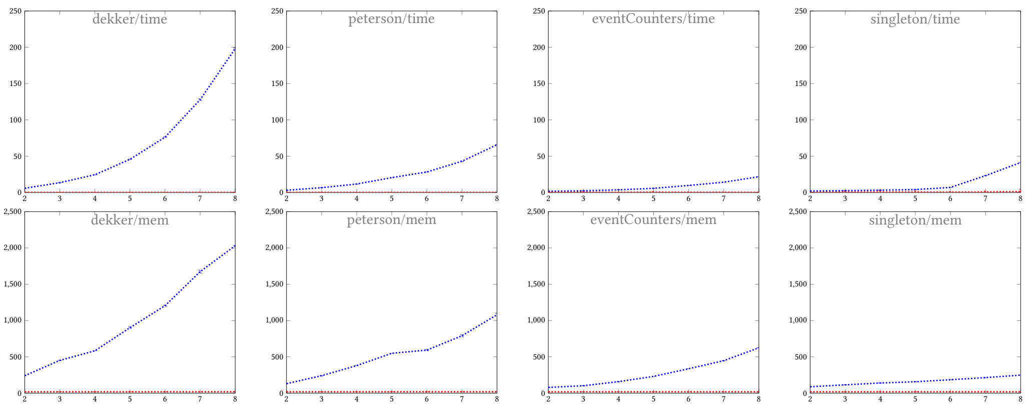


Fig. 7. Time (s) and memory consumption (MB) for Rocker and RSan per number of threads

In addition to the non-robust benchmarks in Fig. 7, we also tested non-robust versions of Lamport's fast mutex algorithm [Lamport 1987] and of sequence lock [Boehm 2012], which starkly highlight the differences between the two tools. On the one extreme, in the `lamport` benchmark, RSan is able to immediately find a robustness violation, while Rocker runs out of memory even for just three threads. On the other extreme, however, in the `seqlock` benchmark, RSan is unable to find the robustness violation within the allocated time budget. Indeed, in this benchmark the violation requires an intricate scheduling to manifest, but the operating system does not context-switch in the appropriate places. Although employing a more sophisticated scheduling algorithm in RSan would help in such cases, we leave such an implementation for future work.

Overall—and despite the synthetic/non-representative nature of the tests—we observe that RSan scales to more threads, and consumes much less memory than Rocker. More importantly, we also observe that RSan finds robustness violations quickly, an observation also confirmed by our realistic tests (see §6.2), and which is useful for determining whether a given program is robust.

Let us now consider some robust programs. We constructed robust versions of the benchmarks in Fig. 7, and measure how much memory the two tools used, as well as how many executions RSan managed to explore until Rocker established robustness. The results followed trends similar to the non-robust case. RSan used a constant amount of RAM of ~18.5MB while Rocker required from ~90MB (for small tests) up to ~2GB, at which point it exceeded our memory limit. In terms of time, RSan needed ~0.03s to execute one run of any benchmark, which means it was able to run up to thousands of runs for most benchmark in the time Rocker required to establish robustness (this varied from ~1s to ~5m).

We note that all benchmarks we used only access a few memory locations. Since Rocker cannot handle even small synthetic tests, it is unlikely to scale to realistic benchmarks with many locations.

6.2 Real-World Case Studies

We used RSan to establish robustness of large, non-trivial concurrent programs. Whenever RSan found a violation, we added an SC fence around the offending instructions or strengthened their access modes, and reran RSan until no new violations were reported within 100 iterations. Since RSan typically found robustness violations after 1–2 iterations, we considered such a limit to be appropriate. In all tests, we were able to ran RSan without modifying the underlying code.

Chase-Lev Deque. Chase-Lev deque [Chase and Lev 2005] is a popular wait-free work-stealing queue. Lê et al. [2013] carefully studied its implementation under weak memory (and it was also studied in the context of a Rust library[5]), while Margalit and Lahav [2021] proved an idealized implementation robust, by adding a few more fences and strengthening some access modes.

For our first case study, we checked whether RSan could prove robustness of an implementation that cannot be verified with state-of-the-art tools like Rocker. To that end, we obtained a popular Chase-Lev implementation[6] (143 stars on Github), and checked its robustness. All in all, RSan discovered 15 robustness violations, which we fixed by inserting 11 fences and promoting 4 relaxed accesses to release/acquire. Each violation was discovered with about a single iteration on average, confirming our observation in §6.1 that RSan identifies robustness violations quickly. Interestingly, with this approach the obtained (minimal) set of SC fences to be added to the code to establish robustness is different from prior work . While not as efficient, RSan's fence set was determined with minimal understanding of the code. Fixing the robustness violations caused a 1.13x slowdown in RSan compared to the initial non-robust case, which we consider fair considering the number of fences inserted and the overhead they incur for maintaining the instrumentation. Finally, we compare the cost of establishing robustness using RSan to what one would get when naively strengthening *all* atomic accesses to SC, which on certain architectures entail a full memory barrier. For the case of Chase-Lev, the latter would lead to 28% more barriers in the compiled code.

VSync Library. VSync[7] is a concurrent library comprising many different locking and concurrent-data-structure implementations, which were verified under weak memory consistency using bounded model checking [Oberhauser et al. 2021]. In this case study, we wanted to see whether these data structures are also robust and, if not, what is the cost of making them robust.

We took all of VSync's concurrent data structures and checked their robustness. RSan discovered 0.75 robustness violations per test on average, and each violation manifested within about 1.2 iterations. Fixing the violations required 38 extra SC fences in 15 files (data structures and their dependencies), which, depending on the application, may be acceptable given the resulting simplicity in reasoning. Similarly to our first case study above, fixing the robustness violations incurred a 1.3x slowdown in RSan compared to the non-robust case. In this case, however, establishing robustness by promoting all atomic accesses to SC would lead to 1,400% more fences on average (26% in the best case).

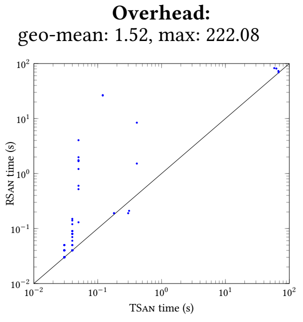
Mimalloc Allocator. `mimalloc`[8] is a memory allocator by Microsoft that heavily uses weak memory atomics to achieve good performance. `mimalloc` has over 10.6k stars on Github and comprises about 12KLoc lines of C code, rendering it our largest case study. We used `mimalloc`'s standard stress testing infrastructure for our client code, which uses 25 threads with a 25% load-per-thread workload, and creates over 60k distinct atomic locations, leading to very large LCs, and thus increased monitoring cost.

RSan identified 20 distinct robustness violations which we repaired one by one. Even though each robustness violation was identified using a single iteration, the time these iterations required varied. The reason for the varying time is the fact that we aborted when encountering robustness violations (thereby not running the whole program), as well as the overhead induced by the added fences. Achieving robustness by promoting all accesses to SC would lead to 300% more fences.

[5]https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md (accessed November 2024)
[6]https://github.com/ConorWilliams/ConcurrentDeque/ (accessed November 2024)
[7]https://github.com/open-s4c/libvsync/ (accessed November 2024)
[8]https://github.com/microsoft/mimalloc (accessed November 2024)

6.3 Overhead over TSan

Finally, we measure the overhead that RSan induces over TSan (using TSan as is, without its adaptions mentioned in §5), which reports 2-20x overhead in execution time on top of non-instrumented execution for typical programs [ThreadSanitizer 2020]. To do so, we ran RSan and TSan on the tests of §6.1 and §6.2, as well as on a large number of automatically generated benchmarks containing a varying number of (randomly generated) memory accesses (154 tests in total). Although the latter set is not representative of real-world workloads, it was designed to bring RSan to its limits, so that we can measure its overhead over TSan in a "worst-case" scenario.



Our results are summarized on the right; on average RSan incurs a 1.52x overhead on top of TSan. This overhead mostly occurs in tests with large thread sizes, and is attributed to the number of memory locations used (more locations imply larger location clocks), and the number of SC fences present. Indeed, while TSan simply executes fences with no instrumentation, RSan has to instrument SC fences, and following the RC20 memory model, it models them as RMWs to an otherwise unused location. Thus, SC fences in RSan contend for the same lock, which introduces a performance penalty.

In certain extreme cases, this penalty might be substantial: e.g., for `mimalloc` (see §6.2), we observe that RSan is 222 times slower than TSan. This is because `mimalloc` has a lot of fences (we used them to make it robust), large threads, and a lot of memory locations: it is a memory allocator, and its stress tests allocate a lot of memory, thereby making RSan's LCs explode. Given that (a) such cases were uncommon in our evaluation, (b) the "push-button" nature of RSan, and (c) the fact that we were able to run RSan on real-world programs, we consider RSan's overall overhead acceptable, and leave improvements to our treatment of fences and LCs as future work.

7 Related and Future Work

Robustness Verification. The robustness of programs against relaxed memory models presents a compelling correctness criterion, enabling the reduction of reasoning and verification under complex, unintuitive models to simpler reasoning that assumes sequential consistency (SC). Robustness with respect to various hardware and programming language memory models has been extensively studied in prior work, such as [Abdulla et al. 2015b,a; Alglave et al. 2017; Alglave and Maranget 2011; Beillahi et al. 2019a,b; Bouajjani et al. 2013, 2018, 2011; Burckhardt et al. 2007; Burckhardt and Musuvathi 2008; Burnim et al. 2011; Derevenetc and Meyer 2014; Lahav and Margalit 2019; Linden and Wolper 2011, 2013; Liu et al. 2012; Margalit and Lahav 2021; Nagar et al. 2024]. Some of these studies present decision procedures, often with PSPACE complexity, similar to the complexity of verifying finite-state concurrent programs under SC. Others propose practical over-approximations of robustness, offering guidance to programmers (and tools) on where to insert memory fences or strengthen access modes. To our knowledge, all existing research has focused on the static verification of robustness, and our paper is the first to propose a dynamic approach.

Enforcing Sequential Consistency. Since weak memory models are inherently complex, researchers have investigated whether their performance benefits justify the added complexity, and explored enforcing SC at the compiler level to simplify software development [Liu et al. 2017, 2021; Marino et al. 2011]. Our focus on robustness stems from a similar perspective. However, as shown in §6, ensuring robustness in specific programs requires significantly fewer fences than enforcing SC through general compiler mappings. Evaluating the performance overhead of making programs robust is left for future work.

Dynamic Race Detection. Dynamic verification of concurrent programs primarily focuses on race detection, as data races are widely regarded as one of the most frequent sources of bugs in concurrent systems. Several tools have been developed and employed for dynamic data race detection, including Eraser [Savage et al. 1997], Djit$^+$ [Pozniansky and Schuster 2007], Helgrind$^+$ [Jannesari et al. 2009], ThreadSanitizer (TSan) [Serebryany and Iskhodzhanov 2009; ThreadSanitizer 2020], and FastTrack [Flanagan and Freund 2009] (see [Yu et al. 2017] for a comparative study). More theoretical work has focused on developing more efficient algorithms to predict additional races without increasing the time and memory overheads (see, e.g., [Kini et al. 2017; Mathur and Pavlogiannis 2022]). None of these dynamic methods supports C11 atomics. Our work extends TSan, which has only limited best-effort support for C11 and is impacted by the observer effect, as discussed in §1.

TSan11. Lidbury and Donaldson [2017] redesigned TSan's race detection algorithm to adapt it for C11. Their tool, TSan11, detects races caused by weak memory behaviors. However, this comes at the cost of significantly more complex and heavyweight instrumentation compared to other race detectors. Specifically, to overcome the observer effect, TSan11 employs software store buffers to track the stores to each memory location during execution. To allow non-SC behaviors, when a read access occurs, TSan11 randomly selects a write from the store buffers to provide the value for the read, subject to constraints that ensure consistency. This approach makes TSan11 resemble random testing rather than a pure dynamic analysis. A further development incorporates controlled randomized scheduling into TSan11 [Lidbury and Donaldson 2019].

Completeness is compromised in TSan11. First, the software store buffers have to be bounded in practice. Second, to maintain manageable complexity TSan11 assumes a memory model stronger than C11 that precludes certain weak behaviors obtained by reordering of consecutive writes:

Example 7.1. Consider the following test (known as 2+2W):

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

When using atomic writes with weak memory orderings (relaxed or even release), C11 permits the outcome $a = b = 1$. This behavior is observable on ARM multiprocessors, which may reorder writes to different addresses. However, TSan11 [Lidbury and Donaldson 2017] would not detect any race that depends on this weak behavior. The tool's algorithm assumes that the "modification order" must align with the execution order (i.e., acyclicity of $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$ rather than of $\mathtt{po} \cup \mathtt{rf}$ as we assume in our work)—a condition that excludes the weak behavior in this example.

C11Tester. A more recent tool, C11Tester by Luo and Demsky [2021], also implements random exploration for detecting data races. C11Tester's instrumentation more accurately models the C11 memory model, particularly by allowing cycles in $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$, which means it correctly handles cases like the one in Ex. 7.1. However, to achieve this, C11Tester must maintain detailed information about the execution graph at each step, resulting in significant memory usage, which grows with the number of write events executed. In contrast, the memory consumption of our approach (which targets a different property) is practically independent of the length of the execution.

Example 7.2. To measure C11Tester's memory consumption, we ran C11Tester on MP from Fig. 1 with 10^N writes to a single (irrelevant) variable in the producer thread. While the program without instrumentation requires 16KB (irrespective of N), C11Tester consumed 26KB for $N = 1$ and 6.3GB for $N = 7$, demonstrating that it requires memory linear in the size of the explored execution graph. (In contrast, RSan requires 19MB regardless of N.)

Model Checking. Stateless model checkers, like Nidhugg [Abdulla et al. 2024, 2019], RCMC [Kokologiannakis et al. 2017], and GenMC [Kokologiannakis et al. 2019], are carefully designed to efficiently explore all possible executions of a (loop-free) program. While these tools can be adapted for random exploration (with GenMC already having a preliminary version for this purpose [Kokologiannakis et al. 2024]), the memory usage per execution remains proportional to the length of the execution.

Testing. Probabilistic concurrency testing [Burckhardt et al. 2010], a randomized testing algorithm that offers theoretical guarantees on the probability of detecting bugs, was applied to the C11 memory model by Gao et al. [2023]. Broadly speaking, this approach involves identifying an appropriate notion of *bug depth* and developing an execution sampling algorithm capable of detecting all bugs up to a certain depth with some probability. Gao et al. [2023] proposed using the number of reads-from edges from one thread to another as the bug depth notion for weak memory programs. However, their sampling algorithm fails to capture executions with cycles in $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$, as the one in Ex. 7.1. Our current approach does not directly control the scheduler, relying instead on the OS for all scheduling decisions. A promising direction for future work is to add such control and incorporate probabilistic concurrency testing of robustness, specifically targeting violations that arise only under particular interleavings (such as those in `seqlock` discussed in §6.1).

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant numbers 814/22 and 2117/23).

Data-Availability Statement

The artifact is available at https://doi.org/10.5281/zenodo.15002567.

References

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. 2024. Parsimonious optimal dynamic partial order reduction. In *CAV*. Springer Nature Switzerland, Cham, 19–43. doi:10.1007/978-3-031-65630-9_2

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, Article 150 (Oct. 2019), 29 pages. doi:10.1145/3360576

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015b. Precise and sound automatic fence insertion procedure under PSO. In *NETYS*. Springer International Publishing, Cham, 32–47. doi:10.1007/978-3-319-26850-7_3

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015a. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*. Springer-Verlag, New York, 308–332. doi:10.1007/978-3-662-46669-8_13

Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't sit on the fence: a static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (May 2017), 38 pages. doi:10.1145/2994593

Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 50–66. doi:10.1007/978-3-642-22110-1_6

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 (July 2014), 74 pages. doi:10.1145/2627752

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, New York, 55–66. doi:10.1145/1925844.1926394

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019a. Checking robustness against snapshot isolation. In *CAV*. Springer International Publishing, Cham, 286–304. doi:10.1007/978-3-030-25543-5_17

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019b. Robustness against transactional causal consistency. In *CONCUR*, Vol. 140. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:18. doi:10.4230/LIPIcs.CONCUR.2019.30

Hans-J. Boehm. 2012. Can Seqlocks get along with programming language memory models?. In *MSPC*. ACM, New York, 12–20. doi:10.1145/2247684.2247688

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. ACM, New York, NY, USA, 68–78. doi:10.1145/1375581.1375591

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: proportional detection of data races. In *PLDI*. ACM, New York, NY, USA, 255–268. doi:10.1145/1806596.1806626

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 533–553. doi:10.1007/978-3-642-37036-6_29

Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning about TSO programs using reduction and abstraction. In *CAV*. Springer, Cham, 336–353. doi:10.1007/978-3-319-96142-2_21

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP*. Springer, Berlin, Heidelberg, 428–440. doi:10.1007/978-3-642-22012-8_34

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*. ACM, New York, 12–21. doi:10.1145/1250734.1250737

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*. ACM, New York, NY, USA, 167–178. doi:10.1145/1736020.1736040

Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 107–120. doi:10.1007/978-3-540-70545-1_12

Jabob Burnim, Koushik Sen, and Christos Stergiou. 2011. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*. Springer, Berlin, Heidelberg, 11–25. doi:10.1007/978-3-642-19835-9_3

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA*. ACM, New York, NY, USA, 21–28. doi:10.1145/1073970.1073974

Egor Derevenetc and Roland Meyer. 2014. Robustness against Power is PSpace-complete. In *ICALP*. Springer, Berlin, Heidelberg, 158–170. doi:10.1007/978-3-662-43951-7_14

Edsger Wybe Dijkstra. 1965. *Cooperating sequential processes, Technical Report EWD-123*. Technical Report. http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, New York, NY, USA, 121–133. doi:10.1145/1542476.1542490

Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic concurrency testing for weak memory programs. In *ASPLOS*. ACM, New York, NY, USA, 603–616. doi:10.1145/3575693.3575729

Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295. doi:10.1109/32.588521

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

ISO/IEC 9899:2011. 2011. Programming Language C.

Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. 2009. Helgrind+: An efficient dynamic race detector. In *IPDPS*. IEEE Computer Society, USA, 1–13. doi:10.1109/IPDPS.2009.5160998

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

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *PLDI*. ACM, New York, NY, USA, 157–170. doi:10.1145/3062341.3062374

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. doi:10.1145/3158105

Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. 2024. Enhancing GenMC's usability and performance. In *TACAS*. Springer Nature Switzerland, Cham, 66–84. doi:10.1007/978-3-031-57249-4_4

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *PLDI*. ACM, New York, NY, USA, 96–110. doi:10.1145/3314221.3314609

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM, New York, 649–662. doi:10.1145/2837614.2837643

Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI*. ACM, New York, NY, USA, 126–141. doi:10.1145/3314221.3314604

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

Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 1–11. doi:10.1145/7351.7352

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *PPoPP*. ACM, New York, NY, USA, 69–80. doi:10.1145/2442516.2442524

Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *POPL*. ACM, New York, NY, USA, 443–457. doi:10.1145/3009837.3009857

Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse record and replay with controlled scheduling. In *PLDI*. ACM, New York, NY, USA, 576–593. doi:10.1145/3314221.3314635

Alexander Linden and Pierre Wolper. 2011. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*. Springer-Verlag, Berlin, Heidelberg, 144–160. doi:10.1007/978-3-642-22306-8_10

Alexander Linden and Pierre Wolper. 2013. A verification-based approach to memory fence insertion in PSO memory systems. In *TACAS*. Springer-Verlag, Berlin, Heidelberg, 339–353. doi:10.1007/978-3-642-36742-7_24

Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. In *PLDI*. ACM, New York, 429–440. doi:10.1145/2254064.2254115

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A volatile-by-default JVM for server applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 49 (Oct. 2017), 25 pages. doi:10.1145/3133873

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2021. Safe-by-default concurrency for modern programming languages. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 10 (Sept. 2021), 50 pages. doi:10.1145/3462206

Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In *ASPLOS*. ACM, New York, NY, USA, 630–646. doi:10.1145/3445814.3446711

William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. 2017. Verifying dynamic race detection. In *CPP*. ACM, New York, NY, USA, 151–163. doi:10.1145/3018610.3018611

Roy Margalit, Michalis Kokologiannakis, Shachar Itzhaky, and Ori Lahav. 2025. Dynamic robustness verification against weak memory (extended version). arXiv:2504.15036 [cs.PL] https://arxiv.org/abs/2504.15036

Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a C11-style memory model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (Jan. 2021), 33 pages. doi:10.1145/3434285

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *PLDI*. ACM, New York, NY, USA, 199–210. doi:10.1145/1993498.1993522

Umang Mathur and Andreas Pavlogiannis. 2022. Dynamic data race prediction: fundamentals, theory, and practice (tutorial). In *ESEC/FSE*. ACM, New York, NY, USA, 1820. doi:10.1145/3540250.3569450

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The complexity of dynamic data race prediction. In *LICS*. ACM, New York, NY, USA, 713–727. doi:10.1145/3373718.3394783

Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*. North-Holland / Elsevier, 215–226.

Kartik Nagar, Anmol Sahoo, Romit Roy Chowdhury, and Suresh Jagannathan. 2024. Automated robustness cerification of concurrent data structure libraries against relaxed memory models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 362 (Oct. 2024), 28 pages. doi:10.1145/3689802

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 *ASPLOS*. ACM, New York, NY, USA, 530–545. doi:10.1145/3445814.3446748

Andreas Pavlogiannis. 2020. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.* 4, POPL (2020), 17:1–17:29. doi:10.1145/3371085

Gary L. Peterson. 1981. Myths about the mutual exclusion problem. *Inform. Process. Lett.* 12 (1981), 115–116.

Eli Pozniansky and Assaf Schuster. 2007. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (March 2007), 327–340.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (nov 1997), 391–411. doi:10.1145/265924.265927

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *WBIA*. ACM, New York, NY, USA, 62–71. doi:10.1145/1791194.1791203

ThreadSanitizer 2020. ThreadSanitizer manual. Retrieved March, 2024 from https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Zhen Yu, Zhen Yang, Xiaohong Su, and Peijun Ma. 2017. Evaluation and comparison of ten data race detection techniques. *International Journal of High Performance Computing and Networking* 10, 4-5 (2017), 279–288. doi:10.1504/IJHPCN.2017.086532

Received 2024-11-15; accepted 2025-03-06