COMMUTATIVITY-AWARE RUNTIME VERIFICATION FOR CONCURRENT PROGRAMS

A Thesis

by

YAHUI SUN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2021

Major Subject: Computer Science

ABSTRACT

Concurrent programs are notoriously difficult to write correctly, as scheduling nondeterminism can introduce subtle errors that are hard to detect and debug. Data races and order violations are two most common types of concurrency errors, which can lead to crashes, data corruptions, and other unexpected errors in shared-memory programs.

Considerable effort has been made towards developing effective program analysis and verification techniques for concurrency bug detection. Most notably, *runtime verification* aims to infer the presence of concurrency bugs from the execution traces and has been shown very effective in practice.

This thesis first develops an efficient runtime verification technique to detect unseen order violations from the observed execution. Our technique novelly extends existing *predictive analyses* that construct partial ordering over trace events in a streaming fashion, reporting unordered pairs of conflicting events as potential order violations. Unlike existing predictive analyses, our technique aims to detect a wider class of concurrency bugs beyond the traditional races. We introduce the notion of *commutativity order violations.* A commutativity order violation occurs in the input trace σ if there is a witness σ^* in which two non-commutative actions appear in the reversed order. This broad definition captures *both* racy and non-racy interaction at the library interface and thus *strictly subsumes* the standard definition of predictable races. For example, an order violation involving two conflicting actions on the same lock is not a race due to the lock synchronization.

We implemented our algorithm into a tool called OVPredict. To address the performance bottleneck of existing predictive analyses, we further propose a space-efficient shadow word representation for tracking the ordering between conflicting critical sections. Our experiments on several real-world large Go and C++ applications show that our analysis detects more order violations than ThreadSanitizer, an industrial-strength race detector, and scales to traces with billions of events. OVPredict revealed several previously unknown order violations in Kubernetes and CockroachDB.

The second focus of this thesis is to develop an efficient stateless model checking algorithm for concurrent programs that use linearizable commutative data structures. We present NCMC, a new stateless model checking algorithm that exploits semantic commutativity between method invocations in concurrent programs. Unlike most previous approaches that capture independence at the *instruction level*, our approach reasons about independence at both *instruction* and *semantics* levels. We introduce the notion of *semantic commutativity equivalence (SC-equivalence)*, a coarser equivalence than the ones characterized by partial orders over events at the instruction level. Underpinned by SC-equivalence, our algorithm uses a commutativity specification to identify noncommutative operations in each exploration, and avoids exploring executions that do not cover any new abstract state of the program. Our algorithm is sound and complete at the semantics level with respect to a given commutativity specification.

Our work addresses real-world challenges in runtime verification of concurrent programs by incorporating commutativity reasoning in both predictive analysis and model checking to improve coverage and scalability. The insights of this thesis are also potentially applicable to the development of predictive analysis techniques that target a broader class of concurrency bugs such as deadlocks.

DEDICATION

To my family.

ACKNOWLEDGMENTS

Throughout my graduate study at Texas A&M, my advisor Dr. Jeff Huang consistently and tirelessly believed in me, mentored me, and advocated me. I'm eternally grateful for his guidance and support.

Together with my advisor, Dr. Jennifer Welch and Dr. Paul Gratz served on my thesis committee and provided valuable help and feedback. Dr. Jennifer Welch also helped me in several other ways; her class on distributed computing broadened my horizon and her support has been crucial to my professional success.

I learned a lot about teaching, life and how to be a good person from Dr. Aakash Tyagi. It was a true pleasure working with him as a teaching assistant for his class.

It has been a great journey to work with all the fantastic lab mates (too numerous to list) in the Origin Oriented Programming Lab at Texas A&M.

José Calderón mentored me and helped me tremendously in COVID lockdowns. I'm grateful to Dale and Merit Weeks for all the meals, love and support. Many thanks also to my roommate and study partner Daochen for having fun and discussing new research ideas together.

I thank my parents for their love and support throughout my life. Last but not least, I would like to thank my girlfriend Hanzi for accompanying me through the ups and downs in my graduate school.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors Jeff Huang, Jennifer Welch of the Department of Computer Science and Engineering, and Professor Paul Gratz of the Department of Electrical and Computer Engineering.

Funding Sources

This work is funded by an NSF award.

NOMENCLATURE

COV	Commutativity order violation
PEG	Plain execution graphs
SEG	Semantic execution graphs
SC-equivalence	Semantic commutativity equivalence
SMC	Stateless Model Checking
SMT	Satisfiability Modulo Theories
TSAN	ThreadSanitizer
VC	Vector clocks
CCS	Conflicting critical section order
COMM	Communication order
DC	Doesn't-commute relation
EDC	Extended-doesn't-commute relation
HB	Happens-before relation
IO	Invocation order
NC	Non-commutative causal order
PO	Program order
RB	Reads-before relation
RF	Reads-from relation
SSH	Semantic-happens-before relation
WCP	Weak-causal-precedence relation
WDC	Weak-doesn't-commute relation
WO	Write order

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

Concurrent programs are prevalent in modern software systems. However, writing correct concurrent programs has remained a huge challenge, as scheduling nondeterminism introduces subtle errors that are hard to detect and debug. Concurrency bugs can cause data corruption, hangs, and crashes [3, 4, 5, 6, 7], incurring significant cost [8] in software development and maintenance.

A large portion of concurrency bugs are due to *order violations* [3], in which the expected ordering between two actions is not enforced in the execution. Order violations can be seen as a generalization of data races. It is not uncommon for a simple order violation error to escape rigorous testing and manifest in production.

Detecting concurrency bugs has received great attention in recent years [7, 9, 10, 11, 12, 13, 14, 15]. The major challenge of concurrency bug detection is the huge number of interleavings of concurrent programs (typically exponential in the program size). Subtle concurrency errors can hide in rare thread interleavings.

One effective approach to mitigate this is *predictive analysis*. Predictive analysis aims to infer potential concurrency errors from the observed execution [16, 17, 18, 10, 19, 20, 21, 22, 23, 24, 25]. Intuitively, predictive analysis considers an equivalent class of correct reorderings of the observed execution. Thus it is capable of predicting unseen concurrency errors knowable from the input trace and achieves higher coverage than purely evidence based methods.

1.1 Motivation and Problem Statement

Existing predictive analyses have limited coverage because they reason about dependency of events in an execution at the level of instructions, and miss the opportunity to capture a larger equivalent class of traces at the library interface level. As modern software increasingly embraces libraries as its building blocks, capturing dependency between library method invocations can significantly improve the power of predictive analyses.

1.1.1 Problem Statement.

To efficiently detect order violations in large shared-memory multi-threaded programs requires robust on-the-fly dynamic analyses that infer unseen order violation from the observed trace. Furthermore, to cover the entire state space of the program without redundancy, each semantically equivalent class of executions should be visited at most once. *Existing predictive analyses and stateless model checking approaches miss the opportunity to consider a larger equivalent class of executions characterized by dependency at the level of library interface.* Efficient dynamic program analysis and stateless model checking approach that is able to exploit commutativity of method invocations beyond basic read-write conflict are yet to be developed.

1.1.2 Commutativity Order Violations

First, this work specifically deals with a class of concurrency errors that we call *commutativity order violations*. Libraries and classes specify an informal contract on the expected ordering between method invocations. A commutativity order violation occurs when such a contract is violated. To illustrate the concept, consider the following concurrent program:

$$\begin{array}{l||l} \texttt{lock}(m) & \texttt{lock}(m) \\ *x = 1 & \texttt{delete}\ x \\ \texttt{unlock}(m) & \texttt{unlock}(m) \end{array} \qquad \text{(Example 1)}$$

where m is a lock and x is a pointer to an integer allocated in the heap. There are two possible interleavings. First, if the right thread is executed before the left thread, we have a use-after-free (UAF) error in which a write to `x` happens after it is freed. Second, if the left thread is first executed, we have a commutativity order violation between `*x = 1` and `delete x`: the two operations do not commute *and* their order can be reversed in an alternative execution. Such concurrent UAF errors are typically severe security vulnerabilities and are very common in web browsers. Existing *evidence-based* memory checkers [26, 27, 28] are effective in finding the UAF in the first interleaving, but fail to *infer* the existence of the error when observing the second interleaving.

Missing Order Violations. Existing dynamic race detection techniques have limited capability in detecting such commutativity order violations. Race detection based techniques that model library operations as read-write accesses [29, 30, 31, 32] *inherently* miss the error in Example 1, since the two operations are (ill-) protected by the same lock. For example, in happens-before (`HB`) analysis, the two critical sections will always be ordered by an `HB` edge (regardless of which thread is first executed), and thus hiding the UAF. We refer to such order violations where the two operations are protected by the same lock as *non-racy* bugs in this thesis.

Active delay-injection approaches [15, 33, 14, 34, 35, 13] also fail to effectively expose such non-racy commutativity order violations. To see why, consider a tool that injects delay upon `*x = 1` in Example 1 in hope of letting a `delete x` happen first. However, delaying `*x = 1` blocks the other thread from acquiring lock m and prevents `delete x` from happening. Even in the racy case where the two operations are not protected by the same lock, delay-injection techniques have high overhead in practice, since the number of memory accesses is typically much larger than free operations.

Static analysis has been used to target order violation errors such as concurrent UAF errors, but typically reports a large number of false positives and is not scalable to large real-world applications [36]. SMT-based predictive analyses [7, 16, 10, 37] can potentially detect all knowable commutativity order violations from a trace by encoding the feasibility constraints of order violations into SMT formulas, but require expensive offline analysis on the logged trace and cannot scale to full execution traces.

Of course, not all commutativity order violations are harmful, however the presence of a commutativity order violation may indicate undesirable interference.

Commutativity Order Violation Prediction. In this work, we introduce the concept of a *commutativity order violation* (COV) and present an efficient algorithm that *predicts* COV bugs in the observed execution with appropriate formal guarantees. Our approach is based on a novel combination of recent partial order based race prediction techniques [17, 20, 22, 21, 38, 23] and a structural representation obtained from a commutativity specification [39]. Conceptually, the notion of COV can be seen as a generalization of both predictive race detection and commutativity race detection [39] to deal with order violations in the library interface level that do not appear as read-write races.

The *de facto* standard notion of correct reorderings is based on low-level reads and writes — it captures feasible reorderings of a trace that satisfies the thread order and same last writer for all reads (or lack of such a writer) [24, 23, 40]. This misses opportunities for capturing a larger set of feasible reorderings based on *semantics*. We introduce the notion of *extended correct reorderings* to capture conflicting operations at the library API level. This allows us to capture a richer and abstract notion of conflict beyond basic reads and writes. We further extend an existing partial order to track conflicting critical sections on the same lock that contain a pair of non-commutative operations. This extension also applies to all existing predictive analyses that track conflicting critical sections (CCS) and thus is of interest beyond commutativity race prediction. To avoid reporting false non-racy order violations, our analysis OVPredict defers reporting non-racy order violations until the end of critical sections.

1.1.3 Efficient Commutativity Library Interface Handling Stateless Model Checking

Our second contribution is a stateless model checking tool that systematically enumerates non-redundant interleavings based on the notion of dependency at the library interface level. *Stateless model checking* (SMC) is an effective way of checking whether assertions in a multithreaded program are always satisfied, which systematically enumerates all nonequivalent executions of the program and checks each of them individually. SMC performed naively does not scale due to the combinatorial explosion of the number of thread interleavings.

Redundant Executions at the Semantic Level. *Dynamic partial order reduction* (DPOR) techniques cut down some of the redundant explorations by partitioning executions into equivalent classes and exploring exactly one execution per equivalent class [41]. [42] proposes an optimal method for DPOR, and it has also been extended to the settings of weak memory consistency [43, 44]. To further reduce the number of explored executions, [45] propose a combination of unfoldings and DPOR, which achieves optimal reduction under some cases. [46, 47] propose *maximal causality reduction* (MCR), another SMC technique that can in principle explore many fewer states than DPOR. To make SMC faster for memory models with declarative semantics such as C++ concurrency, [48] propose execution graph-based approaches that are shown to be more scalable than prior approaches that explore interleavings.

Despite recent advances in SMC, most approaches are only able to exploit the independence at the *instruction level*, but not at the *semantics level*. For example, consider the following program where N threads concurrently add distinct elements to a *set* object s implemented with a linked list. Suppose that the *add*() operation on s is atomic.

$$\begin{array}{c} s.add(1); \;\Big\|\; \ldots \;\Big\|\; s.add(N); \\ \textbf{assert}(s.size() = N); \end{array} \tag{N-SET-ADD}$$

At the semantics level, the execution order of these *add*() invocations does not matter; all orders yield the same return values for each invocation and produce the same set (albeit not the same linked list). Therefore, exploring only one execution suffices to check the assertion at the end. However, these *add*() invocations do not commute at the instruction level: there exists memory dependence between reads and writes (a.k.a. *reads-from* relation) within these invocations, and their different orders produce different linked lists. Any DPOR algorithms that exploit independence at the instruction level would therefore conservatively conclude that these invocations do not commute. As such, they explore $N!$ possible executions to cover the full state space. These executions may differ in the concrete states of the set s. However, these differences are not observable by the client of the set data structure. Considering all of them is therefore unnecessary.

Existing SMC approaches have not exploited such semantic commutativity between method invocations on data structures. The only SMC algorithm that is able to exploit independence of critical sections at the semantics level is CDPOR [49]. Their approach, however, can only utilize independence at the level of concrete states rather than the abstract states. More discussion of their work can be found in Chapter 5. Furthermore, in the face of the growing complexity of software, exhaustively searching all the concrete states of a program naively is not scalable. It is often more desirable and even necessary to only reason about program behaviors based on the interactions among operations on the library interface. Indeed, as observed in [39], concurrent threads can invoke operations in a way that causes undesirable interference at the library interface level, leading to incorrect program behaviors.

Commutativity-aware Stateless Model Checking. As the second contribution of this work, we introduce the notion of *semantic commutativity equivalence* (SC-equivalence), which is coarser than the ones characterized by partial orders over events at the instruction level. Intuitively, two executions are SC-equivalent if one can be obtained from the other by swapping adjacent commutative operations. Based on this notion, we present NCMC (*non-commutativity model checking*), a new SMC algorithm that is able to effectively exploit semantic commutativity. Our approach uses a commutativity specification[1] on concurrent objects to identify commutative operations in each exploration of the program, and avoids exploring executions that do not cover a new abstract state. One key difference from previous approaches is that our approach covers all *abstract states* rather than *concrete states* of the program. This weaker completeness property, called *semantic completeness*, enables our algorithm to achieve significant reduction on the number of explored executions, while still ensuring that all behaviors of the program at the semantic level are covered.

[1]We assume the commutativity specifications are given by the user. Automatic generation of commutativity specifications has also been studied in the literature [50, 51].

1.2 Contributions and Thesis Organization

The thesis presents the following work divided into chapters:

- We introduce the preliminaries on execution models, existing predictive analyses and corresponding relations, and closely related work is discussed in Chapter 2.
- We present our work OVPredict, a predictive analysis that novelly extends existing race prediction techniques to order violation prediction. OVPredict specifically deals with a generalized form of order violations that we call *commutative order violations*. OVPredict is underpinned by a new partial order called *extended doesn't-commute* (EDC) that generalizes existing partial order based predictive analysis by incorporating dependency between library method invocations. To make OVPredict practical, we further present an optimization for tracking conflicting critical sections (CCS) that enables our tool to be competitive with highly optimized happens-before (HB) analysis (Chapter 3).
- We present a stateless model checking algorithm that can exploit semantically equivalent traces and explores *exponentially fewer* interleavings on programs that use commutative data structures, provided with a correct commutativity specification of the data structure (Chapter 4).
- We discuss related work in Chapter 5.
- We summarize the impact of this work and conclude this thesis in Chapter 6.

2. PRELIMINARIES

This chapter introduces basic notation useful throughout the thesis. The exposition follows other related works in the literature.

2.1 Execution model

Actions. We consider concurrent programs consisting of threads that communicate via shared *objects*. Each object $o \in \mathsf{Obj}$ can be in a set of *abstract* states. That is, we are interested in the abstract states of the object as described by its specification and not in the actual implementation details of the object. For example, the actual state of a set object is the set of values it holds.

We assume that object methods are given by specifying their effects on the shared state. For example, Table 2.1 describes the method effects of a set object. We refer to method invocations as *actions*.

An action $a \in \mathsf{Act}$ is denoted by an expression of the form $o.m(\vec{u})/\vec{v}$ where $o \in \mathsf{Obj}$ is an object, m is a method of o, and $\vec{u}$ and $\vec{v}$ are tuples of concrete arguments and return values that match the signature of m. The effect on the abstract state of every $a \in \mathsf{Act}$ is given by a partial map $(\!|a|\!) \in \mathsf{H} \rightharpoonup \mathsf{H}$. For example, for a set object o, the map $(\!|o.\mathsf{size}()/n|\!)$ is identity on all states in which the set o has size n and undefined otherwise. We regard reads and writes as actions on registers, *e.g.,* $\mathtt{rd}(x)$ is a read on register x.

Commutativity. We say that two invocations commute when independent of their application order, their composed effects are the same. Formally, two actions $a, b \in \mathsf{Act}$ *commute*, denoted by $a \bowtie b$, if and only if $(\!|a|\!) \circ (\!|b|\!) = (\!|b|\!) \circ (\!|a|\!)$.

For example, following Table 2.1, two add actions commute when they add different values, as the two actions modify disjoint parts of the (abstract) state of the object. We assume that actions of different objects always commute. That is, a method invocation on one object does not affect the state of another object.

Method	Effect on the set s
$\texttt{add}(v)/r$	$s \longrightarrow s'$ iff $s' = s \cup \{v\}$ and $r = [v \in s]$
$\texttt{contains}(v)/r$	$s \longrightarrow s'$ iff $s' = s$ and $r = [v \in s]$
$\texttt{size}()/r$	$s \longrightarrow s'$ iff $s' = s$ and $r = \lvert s \rvert$

Table 2.1: Methods for a set s and their effects.

Traces and Events. We work on (execution) *traces* of concurrent programs and assume the sequential consistency memory model. In this setting, a trace σ is a totally ordered list of events. Each event e of σ, written as $\langle t, a \rangle$, denotes that the action a is performed by thread $t \in \mathsf{Tid}$, where Tid denotes the set of thread identifiers. We denote by $\texttt{Event}_\sigma$ the set of events in σ; we also denote events simply by their operation (*e.g.,* $\texttt{rd}(x)$, $\texttt{wr}(x)$, $\texttt{acq}(m)$, and $\texttt{rel}(m)$). We use the helper function $\texttt{act}(e)$ to denote the action of event e.

We require that traces obey lock semantics, *e.g.,* every lock is released by a thread t only if there is an earlier matching acquire event by the t, and a lock is held by at most one thread at a time.

Orders on Traces. We define the *trace order* $\prec^\sigma_{\mathsf{tr}}$ as the total order on events of σ, *i.e.,* $e_1 \prec^\sigma_{\mathsf{tr}} e_2$ iff e_1 occurs before e_2 in σ. We define *program order* (or *thread order*) ($\texttt{PO}$) as a strict partial order that orders events in the same thread. Given two events a, b, we write $a \prec^\sigma_{\texttt{PO}} b$ if a occurs before b in σ and a, b are in the same thread.

Given a trace σ and designated sets of *read* ($\texttt{R}$) and *write* ($\texttt{W}$) events, we define the *reads-from* relation $\texttt{RF} \subseteq (\texttt{Event}_\sigma \cap \texttt{W}) \times (\texttt{Event}_\sigma \cap \texttt{R})$ to be total and functional on its second argument: every read event reads from exactly one write event[1]. Under sequential consistency, a read event e observes the last write event (according to the trace order $\prec^\sigma_{tr}$) e' such that e and e' access the same variable and $e' \prec^\sigma_{tr} e$.

Correct Reorderings. A fundamental challenge in concurrency bug detection is the large number of thread interleavings (typically exponential in the program size). The notion of

[1]In the case where no such write exists in the trace, we let the read event read from the initialization write for convenience.

correct reorderings attempts to alleviate this problem by capturing a more coarse equivalent class of traces other than the input trace. The idea here is to infer order violations that might occur in alternate reorderings of an observed trace, thereby detecting order violations beyond those in just the execution that was observed. The set of allowable reorderings of an observed trace σ is defined in a manner that ensures that races or order violations can be detected agnostic of the program that generated σ in the first place. Such a notion is captured by a *correct reordering* in the literature [23, 24, 40, 20], which we refer to as $\mathtt{PORF}$-reorderings in this work.

Given a relation $\mathtt{r}$, we write $\mathtt{r}^+$ for the transitive closure of $\mathtt{r}$. For a partial order $\mathtt{P}$ over σ, a set of events $S \subseteq \mathsf{Events}_\rho$ is said to be *downward-closed* with respect to P_σ if for every $e, e' \in \mathsf{Events}$, if $e \prec^\sigma_{\mathtt{P}} e'$ and $e' \in S$, then $e \in S$.

A trace ρ is said to be a $\mathtt{PORF}$*-reordering* of trace σ if

(1) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$, and

(2) Events_ρ is downward-closed with respect to $(\mathtt{PO}_\sigma \cup \mathtt{RF}_\sigma)^+$, and further $\mathtt{PO}_\rho \subseteq \mathtt{PO}_\sigma$, $\mathtt{RF}_\rho \subseteq \mathtt{RF}_\sigma$.

Soundness and Completeness. A predictive race detection algorithm is *sound* if given any input trace *tr*, every reported race is a predictable race of *tr* (*i.e.,* absence of false positives). The algorithm is called *complete* if it reports all predictable races of *tr* (*i.e.,* absence of false negatives) [2].

2.2 Existing Partial Orders

We briefly describe the *happens-before* ($\mathtt{HB}$) and the *weak-doesn't-commute* ($\mathtt{WDC}$) partial orders.

Definition 2.2.1 (The $\mathtt{HB}$ partial order)**.** Given a trace σ, $\mathtt{HB}$ is the smallest partial order satisfying the following properties:

[2] We note that these notions are often used in reverse in static analysis and program verification. However, here we align with the terminology used in predictive analyses.

(1) (Synchronization order) Release and acquire events on the same lock are ordered by HB.

(2) (Program order) PO $\subseteq$ HB.

The HB relation is sound but incomplete; it has no false positives but misses predictable races knowable from the input trace. HB analysis can be implemented efficiently using vector clocks [52] and it is one of the most widely used dynamic race detection techniques in practice.

Definition 2.2.2 (The WDC partial order)**.** Given a trace σ, WDC is the smallest partial order satisfying the following properties:

(1) (CCS order) If two critical sections on the same lock contain two conflicting events, then the unlock of the first critical section is ordered by WDC before the second conflicting event.

(2) (Program order) PO $\subseteq$ WDC.

The first rule of WDC is commonly referred to as *conflicting critical section* (CCS) order in the literature. WDC is unsound; it may report false races. In practice, almost all WDC races are true races [23].

WDC **Analysis Details.** Algorithm 1 shows the details of an unoptimized algorithm for WDC analysis [23]. The algorithm computes WDC using vector clocks that represent logical time. A vector clock $\mathtt{C} : \mathsf{Tid} \mapsto \mathsf{Val}$ maps each thread to a nonnegative integer [52]. Operations on vector clocks are pointwise comparison ($\sqsubseteq$) and pointwise join ($\sqcup$):

$$\begin{aligned} \mathtt{C}_1 \sqsubseteq \mathtt{C}_2 \quad &\Longleftrightarrow \quad \forall t.\mathtt{C}_1(t) \leq \mathtt{C}_2(t) \\ \mathtt{C}_1 \sqcup \mathtt{C}_2 \quad &\equiv \quad \lambda t.max(\mathtt{C}_1(t), \mathtt{C}_2(t)) \end{aligned}$$

The algorithm maintains the following analysis state:

Algorithm 1 Unoptimized WDC analysis

```
1: procedure RELEASE$(t, m)$
2:     foreach $x \in R_m$ do $\mathtt{L}^r_{m,x} \leftarrow \mathtt{L}^r_{m,x} \sqcup C_t$                    $\}$
3:     foreach $x \in W_m$ do $\mathtt{L}^w_{m,x} \leftarrow \mathtt{L}^w_{m,x} \sqcup C_t$                    $\}$   ▷ WDC rule (a)
4:     $R_m \leftarrow W_m \leftarrow \emptyset$                                              $\}$   (CCS ordering)
5:     $C_t(t) \leftarrow C_t(t) + 1$                                                          ▷ WDC rule (b)
6: procedure WRITE$(t, x)$                                                                     (PO ordering)
7:     foreach $m \in$ HeldLocks$(t)$ do                                     $\}$
8:         $C_t \leftarrow C_t \sqcup \left(\mathtt{L}^r_{m,x} \sqcup \mathtt{L}^w_{m,x}\right)$              $\}$   ▷ WDC rule (a)
9:         $W_m \leftarrow W_m \cup \{x\}$                                     $\}$   (CCS ordering)
10:    check $W_x \sqsubseteq C_t$
11:    check $R_x \sqsubseteq C_t$
12:    $W_x(t) \leftarrow C_t(t)$
13: procedure READ$(t, x)$
14:    foreach $m \in$ HeldLocks$(t)$ do                                     $\}$
15:        $C_t \leftarrow C_t \sqcup \mathtt{L}^w_{m,x}$                                 $\}$   ▷ WDC rule (a)
16:        $R_m \leftarrow R_m \cup \{x\}$                                     $\}$   (CCS ordering)
17:    check $W_x \sqsubseteq C_t$
18:    $R_x(t) \leftarrow C_t(t)$
```

- a vector clock $\mathtt{C}_t$ for each thread t that represents t's current time;
- vector clocks $\mathtt{R}_x$ and $\mathtt{W}_x$ for each program variable x that represent times of reads and writes, respectively, to x;
- vector clocks $\mathtt{L}^r_{m,x}$ and $\mathtt{L}^w_{m,x}$ that represent the times of critical sections on lock m containing reads and writes, respectively, to x;
- sets $\mathtt{R}_m$ and $\mathtt{W}_m$ of variables read and written, respectively, by each lock m's ongoing critical section (if any).

The algorithm checks for WDC-races by checking for WDC ordering with prior conflicting accesses to x; a failed check indicates a WDC-race (lines 10, 11, and 17). The algorithm updates the logical time of the current thread's last write or read to x (lines 12 and 18).

Performance Cost for Computing CCS. Computing CCS order is one major performance cost in predictive analysis [23]. Chiefly, each access with L locks may acquire L T-dimension vector clocks in the worst case.

OVPredict employs the same epoch optimizations as SmartTrack [23], in which vector clocks that store last-access times for writes and most reads can be replaced with a lightweight representation called *epochs*. To mitigate CCS tracking cost, OVPredict uses a space-efficient algorithm similar to the SmartTrack Optimizations (§3.5).

2.3 Commutativity Specifications

The conditions under which two actions commute are often conveniently specified in the form of logical formulas. This allows formal treatment of various types of conditions.

Commutativity Specifications. Given a suitable logic, a logical commutativity specification for a pair of methods m_1, m_2 of the same object is given by a logical predicate $\varphi_{m_2}^{m_1}$ with its free variables collected into the list $(\vec{x_1}; \vec{x_2})$ so the variables $\vec{x_i}$ match the arguments and returns of m_i. A logical commutativity specification Φ for an object with methods M is a set of method specifications $\varphi_{m_2}^{m_1}(\vec{x_1}; \vec{x_2})$ for each $\{m_1, m_2\} \subseteq \mathsf{M}$.

For a pair of methods m_1, m_2 of the same object, the logical predicate $\varphi_{m_2}^{m_1}$ is *sound* if $\varphi_{m_2}^{m_1}(a, b)$ implies that a and b commute for a pair of actions $a, b \in \mathsf{Act}$. A commutativity specification Φ is sound if every predicate in Φ is sound. In this work, we only consider sound commutativity specifications.

Note that sound commutativity specifications are not necessarily precise. Even though the actions commute, the specification is allowed to say that they do not. Our algorithms require sound commutativity specifications.

Access Point Representation. We next introduce a structure which is useful for capturing commutativity specifications in a way that can be used by a dynamic program analyzer. We refer to these structures as *access point representations* (of the object's commutativity properties).

Definition 2.3.1 (Access point representation)**.** An access point representation for an object $o \in \mathsf{Obj}$ is a tuple $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$, where

1. $\mathcal{X}_o$ is a set of access points. Intuitively, access points capture the micro-action underlying a method invocation.

2. $\eta_o \in \mathsf{Act}_o \to \mathcal{P}(\mathcal{X}_o)$ indicates the finite set of access points touched by each action of the object (Act_o stands for the set of all actions of object o).

3. $\mathcal{C}_o \subseteq \mathcal{X}_o \times \mathcal{X}_o$ is a symmetric binary relation describing which access points conflict.

We now define what it means for an access point representation to precisely match a logical specification. Automatic translation from a logical formula to an equivalent access point representation is studied in [39].

We say that $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$ represents a logical commutativity specification Φ of o iff for all $\varphi^{m_1}_{m_2} \in \Phi$ and all actions a of m_1 and b of m_2, we have:

$$(\eta_o(a) \times \eta_o(b)) \cap \mathcal{C}_o = \varnothing \text{ iff } \varphi^{m_1}_{m_2}(a, b)$$

Therefore, we can now work with an access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$ instead of an equivalent commutativity specification Φ. We utilize access point representations in our commutativity order violation detector (§3.4).

3. OVPREDICT: EFFICIENT COMMUTATIVITY ORDER VIOLATION PREDICTION

3.1 Introduction

First, in this work, we introduce the concept of a *commutativity order violation* (COV) and present an efficient algorithm that *predicts* COV bugs in the observed execution with appropriate formal guarantees. COV is a *sound* notion of order violations that captures both racy and non-racy order violations, and it *strictly subsumes* the standard notion of predictable races.

Next, we introduce the notion of *extended correct reorderings* (EDC) to capture conflicting operations at the library API level. This allows us to capture a richer and abstract notion of conflict beyond basic reads and writes. We further extend an existing partial order to track conflicting critical sections on the same lock that contain a pair of non-commutative operations. EDC is a generalization of the traditional notion of *correct reorderings* (a.k.a. predictable traces) by considering conflicting operations in the library interface rather than the basic reads and writes. This extension also applies to all existing predictive analyses that track conflicting critical sections (CCS) and thus is of interest beyond commutativity race prediction. To avoid reporting false non-racy order violations, our analysis OVPredict defers reporting non-racy order violations until the end of critical sections.

We develop an efficient single-pass algorithm OVPredict that, given an input trace σ, detects whether σ contains a commutativity order violation. OVPredict can be based on existing (not necessarily sound) partial orders such as WDC. Moreover, OVPredict reports no false non-racy order violations even if the underlying partial order is unsound.

Although the underlying predictive analysis of OVPredict performs a single pass on the input trace, it is not suitable for online deployment due to its complexity. One major performance bottleneck is conflicting critical section (CCS) tracking [23]. To mitigate this, we further propose a space-efficient algorithm that stores only compact mutex-access metadata per memory location.

We implemented OVPredict and evaluated it on a number of large Go and C/C++ code-bases, including Kubernetes and Chromium. OVPredict targets concurrent UAF errors in C/C++ and API order violations in Go. OVPredict found 5 previously unknown order violations in the Go projects. 3 of them have already been confirmed and fixed by the developers. On the Chromium UAF benchmarks, OVPredict has 33% higher rates of detecting the UAF compared to ThreadSanitizer, an industrial-strength race detector.

3.2 Overview

In this section we give an overview of our approach as well as the underlying motivation. Full formal details are presented in later sections.

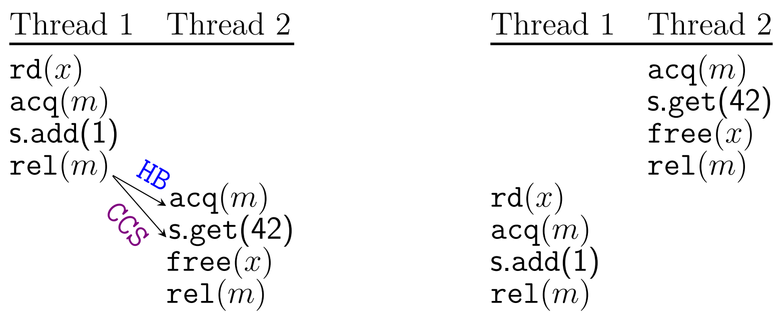
Consider the following program with a set s and a pointer x to an integer:

```
a = *x       ||  lock(m)
lock(m)      ||  if s.get(42) then
s.put(1)     ||     delete x
unlock(m)    ||  unlock(m)
```

(Example 2)

Suppose initially s contains the value 42. The left thread reads the integer pointed by x and then puts the value 1 into the set s. The right thread tests whether s contains 42, and if it is the case, it proceeds to delete x.

Figure 3.1a shows the execution trace where the left thread is executed before the other thread. Here, we use `acq`(m) and `rel`(m) to denote acquire and release lock m respectively. The trace has a predicable COV as Fig. 3.1b demonstrates. The happens-before (HB) analysis clearly misses the predictable COV because of the HB edge between the release-acquire events. Predictive analysis is a promising candidate for detecting predictable COV. We next describe the limitations of existing race prediction techniques and our motivation for our approach.



Thread 1	Thread 2
rd(x)	
acq(m)	
s.add(1)	
rel(m)	
	acq(m)
	s.get(42)
	free(x)
	rel(m)

(a) An execution trace with a predictable non-racy UAF.

Thread 1	Thread 2
	acq(m)
	s.get(42)
	free(x)
	rel(m)
rd(x)	
acq(m)	
s.add(1)	
rel(m)	

(b) The reordered trace revealing the UAF error.

Figure 3.1: The execution in (a) has a predictable UAF as (b) demonstrates.

Existing Predictive Analyses Miss COV. Existing predictive analyses model conflicts at the read-write level. The recently introduced partial orders such as *weak-causally-precedes* (`WCP`), *doesn't-commute* (`DC`) and *weak-doesn't-commute* (`WDC`) all track dependency between critical sections on the same lock with a pair of conflicting memory accesses. Specifically, if two critical sections on the same lock contain conflicting events, *e.g.,* read-write or write-write conflict, the analyses will order the release event of the first critical section before the second conflicting event in the trace. This is referred to as the *conflicting critical sections* (`CCS`) rule in prior work [20, 21, 23, 22].

As depicted in Fig. 3.1a, we have a `CCS` edge between `rel`(m) and s.`get`(1). For brevity, we lift the `CCS` ordering from a release-access edge to a release-invocation edge. Intuitively, the add and get operations on the same set contain low-level conflicting accesses that read and update the internal state of the set.

Due to `CCS` ordering and its composition with the *program order* which orders events in the same thread, the `rd`(x) event is ordered before `free`(x). Thus existing partial order based analysis misses the predictable UAF as depicted in Fig. 3.1b.

Exploiting Commutativity Specifications. To exploit commutativity between s.`add`(1) and s.`contains`(42) in the semantics level, we leverage a commutativity specification for the set interface. For example, the following formula describes when *add* and *get* operations commute:

$$v_1 \neq v_2 \vee (r_1 = \texttt{true} \wedge r_2 = \texttt{true})$$

This formula evaluates to `true` since the two operations operate on different values. Thus the two invocations are not treated as conflicting events by our approach. Our extended CCS rule will keep the two operations unordered, enabling the predictable UAF in Fig. 3.1b to be detected. If the commutativity specification evaluates to `false`, the two operations will be treated as conflicting events and the same CCS rule from prior work applies.

3.3 Commutativity Order Violations

In this section we generalize the definition of correct reordering to capture interaction in the library interface. We will then introduce *commutativity order violations*, the central concept of this thesis.

3.3.1 Generalized Correct Reordering

Rather than considering conflicting events between basic reads and writes, we characterize the correct trace reorderings based on interaction in the library interface. This allows us to exploit the abstract semantics of library interaction and capture a *coarser* class of correct reorderings. Our characterization is inspired by recent work on declarative specification for concurrent libraries [53].

Given a trace σ, we require that the concurrent library specifies the *communication order* (COMM). Intuitively, COMM relates those actions in σ that exchange information. For example, in terms of basic reads and writes, the COMM relation is equivalent to the reads-from (RF) relation, where $(w, r) \in$ COMM denotes that event r reads from the value written by event w. In case of a set library, COMM relates matching update and query operations; *e.g.,* $(s.\texttt{add}(v)/\texttt{true}, s.\texttt{get}(v)/\texttt{true}) \in$ COMM since the `add` and `get` actions 'communicate' through the value v.

We say that the COMM order is *consistent* with the commutativity specification Φ if the actions of every pair of events related by COMM do not commute. Formally, COMM is consistent with Φ if for every $(e_1, e_2) \in \texttt{COMM}$, $\texttt{act}(e_1)$ and $\texttt{act}(e_1)$ do not commute according to Φ. Intuitively, if two events in σ exchange information, they should not be allowed to commute; COMM captures a form of causality between events. Our analysis require that COMM is defined to be consistent with the given commutativity specification.

We next define our generalized trace reordering which we call POCOMM-reordering. POCOMM-reordering is similar to PORF-reordering except it is based on COMM rather than RF.

A trace ρ is said to be a POCOMM-*reordering* of trace σ if

(1) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$, and

(2) Events_ρ is downward-closed with respect to $(\texttt{PO}_\sigma \cup \texttt{COMM}_\sigma)^+$, and further $\texttt{PO}_\rho \subseteq \texttt{PO}_\sigma$, $\texttt{COMM}_\rho \subseteq \texttt{COMM}_\sigma$.

For example, Fig. 3.1b is a correct reordering of Fig. 3.1a.

3.3.2 Commutativity Order Violation

Armed with the generalized notion of correct reorderings, we can now define the notion of *communicatively order violations*, a central concept of our thesis. Intuitively, a *commutativity order violation* (COV) occurs in the input trace σ if there is a POCOMM-reordering of σ in which two non-commutative actions appear in the reversed order.

Definition 3.3.1 (Commutativity order violation)**.** For a trace σ, a pair of events $e_1, e_2 \in \texttt{Event}_\sigma$ with $e_1 \prec^\sigma_{\mathsf{tr}} e_2$ form a *commutativity order violation* if their corresponding actions a, b do not commute and there exists a POCOMM-reordering σ^* such that $e_2 \prec^{\sigma^*}_{\mathsf{tr}} e_1$ (the trace order between e_1, e_2 is flipped).

Our notion of commutativity order violations can be seen as a generalization of both predictable races and commutativity races [39].

3.3.3 Common Order Violations

In this section we present several common types of commutativity order violations that tend to be harmful in practice. Figure 3.2 lists the commutativity specification of 4 types

$$\begin{aligned}
\varphi^{\mathsf{read}}_{\mathsf{free}}, \varphi^{\mathsf{write}}_{\mathsf{free}} &:= \texttt{false}\\
\varphi^{\mathsf{send}}_{\mathsf{close}}, \varphi^{\mathsf{send}}_{\mathsf{send}} &:= \texttt{false}\\
\varphi^{\mathsf{wait}}_{\mathsf{signal}}, \varphi^{\mathsf{wait}}_{\mathsf{broadcast}} &:= \texttt{false}\\
\varphi^{\mathsf{signal}}_{\mathsf{signal}}, \varphi^{\mathsf{signal}}_{\mathsf{broadcast}}, \varphi^{\mathsf{broadcast}}_{\mathsf{broadcast}} &:= \texttt{true}\\
\varphi^{\mathsf{Err}}_{\mathsf{cancel}} &:= \texttt{false}\\
\varphi^{\mathsf{Err}}_{\mathsf{Err}}, \varphi^{\mathsf{cancel}}_{\mathsf{cancel}} &:= \texttt{true}\\
\varphi^{\mathsf{Write}}_{\mathsf{Close}}, \varphi^{\mathsf{Read}}_{\mathsf{Close}}, \varphi^{\mathsf{Read}}_{\mathsf{Write}}, \varphi^{\mathsf{Write}}_{\mathsf{Write}} &:= \texttt{true}
\end{aligned}$$

Figure 3.2: Commutativity specifications for common order violations.

of COV that OVPredict targets. These include (1) use-free interaction in C/C++, and (2) the condition variable, context library and channel operations in Go. Besides UAF which is introduced in §4.1, we aim to detect COV in the following Go libraries based on the survey [4]:

(1) *Sending to a closed Go channel*: Channel send and close operations do not commute.Sending a value to a closed channel causes the program to panic.

(2) *Incorrect usage of condition variables*: Broadcast (or signal) and wait operations on a condition variable do not commute. Broadcast operations that occur before wait events will be lost, potentially causing the Goroutine that waits to starve.

(3) *Incorrect usage of context objects*: Reading the error message of a context object through the `Err` method does not commute with the cancelling of the context object. If the reading of the error occur before it is set by `cancel`, `nil` is returned.

3.4 OVPredict

This section introduces OVPredict, a dynamic analysis that detects both data races and non-racy order violations.

3.4.1 Extended Doesn't-Commute Analysis

In this section we present our partial order, *extended doesn't-commute* (EDC), for commutativity order violation detection. EDC can be seen as a generalization of the existing partial orders such as WDC. EDC captures non-commutativity ordering at the method invocation level instead of the classical read-write level. For simplicity we base our partial order on the WDC partial order. Extending other partial orders for commutativity order violation analysis can be handled in a similar way.

The EDC order is the smallest partial order that satisfies the following properties.

(1) (Extended CCS) If two critical sections on the same lock contain two actions that don't commute, then the first critical section is ordered to the second event.

(2) (Program order) PO $\subseteq$ EDC.

EDC generalizes the WDC relation to capture library method invocations: it considers method invocations that don't commute instead of read-write conflict. Moreover, the same extended CCS order rule applies to *any* predictive analyses that employ the CCS order. These include WCP [20] and DC [21].

EDC **Analysis.** We next present the details of EDC analysis for COV detection, shown in Algorithm 2. EDC analysis consists of two components: (1) the efficient, streaming main procedure for computing the EDC partial order on the input trace (procedure Release, Action, and helper function ConflictingAP), and (2) helper functions for the *deferred check* for potential non-racy COV candidates that decide if a pair of EDC-unordered events on the same lock form a true COV (highlighted procedure CheckCandidate and AddCandidate).

Overview. EDC requires the access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$ obtained from the commutativity specification for the set of objects O as input. The algorithm maintains the following analysis states:

- a vector clock $\mathtt{C}_t$ for each thread t that represent t's current time;

Algorithm 2 EDC analysis

```
1: procedure $\textsc{Release}(t, m)$
2:     foreach $p \in \mathtt{P}_m$ do $\mathtt{L}^r_{m,x} \leftarrow \mathtt{L}_{m,p} \sqcup \mathtt{C}_t$          ▷ EDC rule (a)
3:     $\mathtt{P}_m \leftarrow \varnothing$                                                     (CCS ordering)
4:     $\mathtt{C}_t(t) \leftarrow \mathtt{C}_t(t) + 1$                                          ▷ EDC rule (b)
5:     $\textsc{CheckCandidate}(t, m)$
6: procedure $\textsc{Action}(t, o, a)$                                          (PO ordering)
7:     $AP \leftarrow \textsc{ConflictingAP}(o, a)$
8:     foreach $p \in AP$ do
9:         $\textsc{AddCandidate}(t, p)$
10:    foreach $m \in \text{HeldLocks}(t)$ do                     ⎫
11:        $\mathtt{P}_m \leftarrow \mathtt{P}_m \cup \eta_o(a)$                          ⎪
12:        foreach $p \in AP$ do                                  ⎬          ▷ EDC rule (a)
13:            $\mathtt{C}_t \leftarrow \mathtt{C}_t \sqcup \mathtt{L}_{m,p}$                     ⎭          (CCS ordering)
14:    foreach $p \in \eta_o(a)$ do
15:        $\mathtt{C}_p(t) \leftarrow \mathtt{C}_t(t)$
16: procedure $\textsc{ConflictingAP}(o, a)$
17:     $AP \leftarrow \varnothing$
18:     foreach $p \in \eta_o(a)$ do
19:         foreach $p' \in \mathsf{active}(o) \cap \mathcal{C}_o(p)$ do
20:             $AP \leftarrow AP \cup \{p'\}$
21:     return $AP$
22:
23: procedure $\textsc{CheckCandidate}(t, m)$
24:     foreach $e@u \in \mathsf{Cand}_{t,m}$ do
25:         check $e@u \sqsubseteq \mathtt{C}_t$
26:     $\mathsf{Cand}_{t,m} \leftarrow \varnothing$
27: procedure $\textsc{AddCandidate}(t, p)$
28:     foreach $e@u \in \mathtt{C}_p$ do
29:         if $u \neq t \wedge e@u \not\sqsubseteq \mathtt{C}_t$ then
30:             $M \leftarrow \text{MSet}(e@u, \mathtt{C}_t(u)) \cap \text{HeldLocks}(t)$
31:             if $M = \varnothing$ then
32:                 Report racy COV
33:             else
34:                 foreach $m$ in $M$ do
35:                     append$(\mathsf{Cand}_{t,m}, e@u)$
```

- vector clocks $\mathtt{C}_p$ for each access point $p \in \mathcal{X}_o$. This is analogous to the read and write vector clocks used in classical HB analysis;

- vector clocks $\mathtt{L}_{m,p}$ that represent the release times of critical sections on lock m containing accesses to point p;

- a set $\mathtt{P}_m$ for access points accessed by each lock m's ongoing critical section (if any);

- a candidate list for each pair of thread t and lock m of the form $\langle e_0@t_0, \ldots, e_n@t_n\rangle$, where each $e_i@t_i$ is an epoch representing an access that happen within critical sections under m..

Procedure ConflictingAP. The ConflictingAP helper function iterates over each access point p touched by a and seeks all access points $p' \in \mathsf{active}(o)$ which conflict with p. This is done by computing the intersection of $\mathsf{active}(o)$ and $\mathcal{C}_o(p)$. The conflicting access points are gathered in set AP, which is the return value.

Procedure Action. Upon each action a on object $o \in O$, the procedure Action is called. First, it computes the set of access points touched by a using the helper function ConflictingAP (line 7). Next, it checks the times for each conflicting access point $p \in AP$ for potential order violations though procedure AddCandidate, which we will describe shortly. If the last-access time for point p is not comparable to $\mathtt{C}_t$, the access point will go through a second procedure to check for COV candidates. This is analogous to checking read-write races in classic HB analysis.

Action proceeds by updating state for CCS order (lines 10-13). For each lock held by thread t, it adds all touched access points to $\mathtt{P}_m$, and updates thread clock $\mathtt{C}_t$ by merging $\mathtt{L}_{m,p}$ for each $p \in AP$. Essentially, $\mathtt{C}_t$ acquires the release times of critical sections on m that touch conflicting access points to a. The difference from WDC analysis is that here we use access points here rather than the read and write release times for each variable.

Finally, Action updates the last-access time $\mathtt{C}_p$ for each touched access point p (lines 13-14). It suffices to update only the component in $\mathtt{C}_p$ that corresponds to thread t.

ProcedureRelease. The RELEASE procedure serves two purposes: updating states for CCS order and determining each non-racy COV candidates in $\mathsf{Cand}_{t,m}$. First, it updates $\mathsf{L}_{m,p}$ for each touched $p \in \mathsf{P}_m$, clears P_m, and increments the thread epoch by 1. We next describe how OVPredict determines non-racy COV candidates.

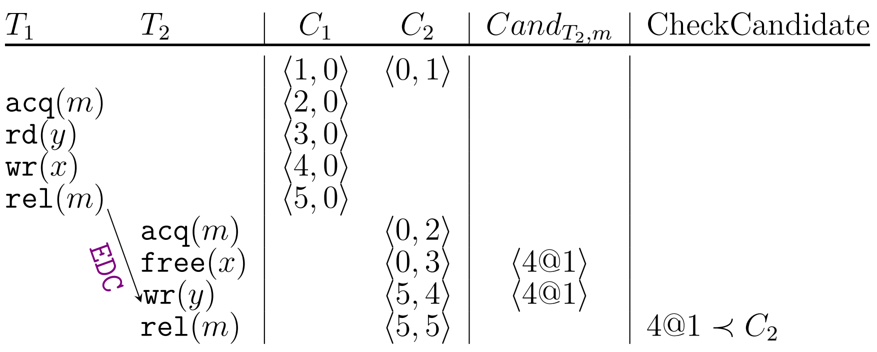
Predicting Non-racy COV. The key insight for deferred check is that a feasible reordering of two non-racy but conflicting simulated accesses in an input trace requires their two entire critical sections be reordered. In other words, there is no causal relation between the two critical sections. This requires the analysis observe all events from the two critical sections before it can decide whether a simulated access pair is a true order violation.

When the second critical section under the same lock ends, *i.e.,* when the lock releases, OVPredict checks if the two entire critical sections are ordered by EDC. If not, OVPredict reports a potential order violation, which is guaranteed to be a true positive.

Purpose of Candidate List $\mathsf{Cand}_{t,m}$. The algorithm uses $\mathsf{Cand}_{t,m}$ to compare the previous access' epoch to the mutex release time of the current thread, in order to determine whether the two critical sections enclosing the two accesses are ordered.

Procedure CheckCandidate. CHECKCANDIDATE adds an epoch to $\mathsf{Cand}_{t,m}$ when it encounters an access point that may form a order violation candidate with a previous access point. The analysis checks two conditions for the current access point: (1) it is EDC-unordered with a previous simulated access point to the same variable, and (2) it holds the same mutex m as the previous access point. When both conditions are satisfied, the previous access point's epoch is appended to the epoch list $\mathsf{Cand}_{t,m}$.

On a release event for mutex m by thread t, the analysis fetches the epoch list of previous access pointes in $Cand_t$ using m. Each epoch in the list is then compared to thread t's current time C_t. If a previous epoch is not ordered before C_t, the two critical sections enclosing the access points must be unordered. OVPredict reports a true OV candidate at this stage. Otherwise, the OV candidate is deemed infeasible. Finally, the m's entry in $Cand_t$ is cleared.



T_1	T_2	C_1	C_2	$Cand_{T_2,m}$	CheckCandidate
		$\langle 1,0\rangle$	$\langle 0,1\rangle$		
`acq`(m)		$\langle 2,0\rangle$			
`rd`(y)		$\langle 3,0\rangle$			
`wr`(x)		$\langle 4,0\rangle$			
`rel`(m)		$\langle 5,0\rangle$			
	`acq`(m)		$\langle 0,2\rangle$		
	`free`(x)		$\langle 0,3\rangle$	$\langle 4@1\rangle$	
	`wr`(y)		$\langle 5,4\rangle$	$\langle 4@1\rangle$	
	`rel`(m)		$\langle 5,5\rangle$		$4@1 \prec C_2$

Figure 3.3: An example for vindicating non-racy UAF candidates.

A Non-racy COV Example. Consider the following UAF-free C++ program with two threads:

```
lock(m)               || lock(m)
if (y ≠ 0)then        || free(x)
  *x = 1              || y = 0
unlock(m)             || unlock(m)
```

(NO-UAF-DEP)

Figure 3.3 shows OVPredict's analysis states for an execution where all events from the left thread finish before the right thread starts. The unlock on `m` on the left thread is EDC-ordered to the read of `y` on the right thread, but not ordered to the free on `x`. Thus, OVPredict adds the two racy epochs $\langle 4@T_1, 3@T_2\rangle$ along with their mutex set intersection $\{$`m`$\}$ to the candidate list D_2 of thread T_2. On the unlock of `m` from T_2, OVPredict compares the vector clock C_2 of T_2 with the first racy epoch $4@T_1$. Since C_2 happens before $4@T_1$, the candidate is vindicated and removed from D_2.

By the EDC rule, the free of x in T_2 is unordered to the write to x in T_1, leading to a potential UAF. However, the UAF is not feasible because the deletion of object x cannot come before the use of object x in an alternative execution. If the critical section on the right hand side executes first, the use event from the left hand side will be unreachable, hence no UAF will happen.

3.5 Optimizing CCS Tracking

This section introduces a space-efficient shadow word representation for CCS tracking. The algorithm which we call *bounded CCS tracking* uses bounded space to store mutex-access metadata and supports

- *Epoch optimizations* are optimizations from prior works [54, 23] that replace vector clocks with epochs for last access metadata. We omit discussion about epoch optimizations in this thesis since OVPredict uses the same epoch optimizations from SmartTrack.

- *Mutex-access metadata optimizations* are novel optimizations that use more memory-efficient and cache-friendly data structures compared to SmartTrack for CCS tracking.

3.5.1 Mutex-access Metadata Optimizations

Tracking CCS is one major source of performance overhead compared to HB analysis [23]. It is expensive and frequently invoked. For each *non-same-epoch* memory access within a critical section[1], previous information that binds an access with its locks held (called *mutex-access information*) must be queried to find a conflicting access that holds the same lock. If found, ordering between the previous mutex release and the current access needs to be established by expensive vector clock merging operations.

While SmartTrack significantly mitigates the performance penalty from CCS tracking, further optimizations can be used reduce memory consumption and maximize cache performance.

OVPredict is our new algorithm that leverages *shadow memory* to achieve lower memory consumption and better cache performance.

Analysis States. OVPredict uses compact data structures called *mutex-access shadow word* to represent a mutex-access pair within critical sections, which can be stored in the same shadow region for each variable.

[1]We assume the application of FastTrack's epoch optimizations.

Definition 3.5.1 (Mutex-access Shadow Words)**.** A mutex-access shadow word is a tuple $\langle t, m, a, w\rangle$, where t is the thread ID, a is the *acquiring count* of mutex m, and w the *write bit* for the access and the mutex, respectively. The acquiring count a is a scalar value that increases by one on each acquire of m. The write bit w for an access indicates whether the access belongs to the write set.

A mutex-access shadow word can be squeezed into an 8-byte word, which can be accessed atomically on a 64-bit processor. More specifically, all elements in a shadow word can be represented using small bit widths. First, the read bits use two bits in total. Second, the total numbers of threads and mutexes created are typically bounded to a small number in practice. Third, by limiting the depth of CCS tracking, the acquiring count of each mutex is also bounded to a small number.

Algorithm 3 OVPredict optimizations for CCS tracking

```
1: procedure Acquire($t, m$)
2:     $N_m \leftarrow N_m + 1$
3: procedure Release($t, m$)
4:     $H_m(N_m) \leftarrow C_t$
5:     CheckCandidate($t, m$)
6: procedure MemAccess($t, x, w$)
7:     foreach $m$ in LocksHeld($t$) do
8:         $W_{\mathsf{new}} \leftarrow \langle t, m, H_m, w\rangle$
9:         UpdateShadow($x, W_{\mathsf{new}}$)
10:    for $\langle e@u, w'\rangle$ in LastAccess($x$) do
11:        check $e@u \prec C_t(t)$                                ▷ Check data race
12: procedure UpdateShadow($x, W_{\mathsf{new}}$)
13:    $\langle t, m, n, w\rangle \leftarrow W_{\mathsf{new}}$
14:    foreach $W_{\mathsf{old}}$ in $S_x$ do
15:        $\langle t', m', n', w'\rangle \leftarrow W_{\mathsf{old}}$
16:        if IsZero($W_{\mathsf{old}}$) then
17:            OverwriteOnce($W_{\mathsf{old}}, W_{\mathsf{new}}$)
18:        else if $(w \vee w') \wedge (m = m')$ then              ▷ Conflicting CS
19:            if $t \neq t'$ then $C_t \leftarrow C_t \sqcup H_{m'}(n')$
20:            if $w$ then OverwriteOnce($W_{\mathsf{old}}, W_{\mathsf{new}}$)
```

Algorithm 3 shows the optimized OVPredict analysis. To track CCS, OVPredict maintains the following additional states:

- An acquiring count N_m for each mutex m;
- A thread-local mutex map M_t for each thread t that maps each mutex m currently held by t to m's acquiring count when t acquires m;
- A vector of vector clocks H_m for each mutex m indexed by the acquiring count. Each vector clock represents the release time of m at that specific acquiring count;
- A mutex-access shadow word list S_x for each program variable x. S_x is an array of mutex-access shadow words, which store the information about last access to x within critical sections.

OVPredict Analysis Overview. OVPredict updates mutex related states N_m and H_m on mutex acquire and release events. On acquiring mutex m, the algorithm increments the acquiring count N_m. On releasing mutex m, the algorithm stores the current thread clock C_t to the N_m-th slot of H_m, and proceeds to check non-racy OV candidates using CheckCandidate in algorithm 2.

On a read or write access, OVPredict creates a new mutex-access shadow word W_{new} for each mutex currently held. OVPredict then proceeds to query each shadow word to x using the UpdateShadow procedural.

Maintaining Mutex-access Metadata. The UpdateShadow procedure queries and updates existing shadow words W_{old} stored in the shadow region of x, using the current shadow word W_{new}. First, if the fetched W_{old} is zero, the analysis stores W_{new} at the same location if it has not yet been stored. The OverwriteOnce procedure ensures that W_{new} is only written once over the old shadow word. If W_{new} has been written, OverwriteOnce overwrites W_{old} with an empty shadow word with value 0. This ensures that no duplicate

	SmartTrack	SwiftTrack
CS list per thread	H_t	None
Ancillary metadata per variable	A_x^w, A_x^r	None
Mutex-access metadata structures	CS lists L_x^r, L_x^w	Shadow words

Table 3.1: Comparing OVPredict and SmartTrack.

shadow words are written to S_x and that space occupied by stale shadow words can be reclaimed.

If W_{old} is not zero, OVPredict checks if the current access conflicts with the previous access by checking that at least one of w and w' is set. If the two accesses are both reads, *i.e.,* not conflicting, no action is needed. Otherwise, OVPredict merges the n'-th vector clock in $H_{m'}$ to the thread clock C_t if the two accesses are from different threads. Next, OVPredict overwrites W_{old} if the current access is a write.

Comparing to SmartTrack. Table 3.1 summarizes the key differences between SwiftTrack and SmartTrack. Unlike SmartTrack which stores mutex-access metadata for reads and writes in separate global-shared linked lists, OVPredict uses mutex-access shadow words to capture both reads or writes within a critical section in a unified way, without resorting to ancillary data structures.

Finally, the OVPredict algorithm is simpler, more understandable and easier to implement than OVPredict.

3.6 Evaluation

We have implemented the OVPredict on top of ThreadSanitizer (TSAN). Our implementation consists of (1) the runtime library built on top of TSAN runtime library in C++, and (2) the static instrumentor that inserts call to the runtime library for Go programs.

Runtime Library. The runtime library is shared for both Go and C/C++ programs. The runtime library implements the optimized version of OVPredict, using shadow memory for mutex-access metadata. For UAF errors in C/C++, OVPredict reuses TSAN's interfaces, but replaces the HB analysis with the new algorithm. OVPredict also reuses TSAN's implementation for the epoch optimizations, which uses shadow memory to store at most 4 last accesses as epochs for each 8-byte application memory.

Static Instrumentor. The static instrumentor is implemented in Go and targets OV bugs for Go libraries. It inserts calls to the runtime library that generate read or write simulated accesses at runtime. We specifically target the five types of OV bugs in Fig. 3.2. OVPredict reuses TSAN's manual instrumentation to the Go channel implementation and WaitGroup synchronization library. For the remaining three types, we run the instrumentor with a list of 16 methods. Among them, 3 are for the condition variable library, 2 are for the context library, and 11 are for the operations on a file descriptor.

We extensively evaluate OVPredict on industry-sized programs, including Kubernetes and Chromium. We present the bug detection results for Go OV bugs and C/C++ UAF bugs in §3.6.2 and §3.6.3. For Go programs, we also compare the effectiveness and performance between an optimized OVPredict with all optimizations applied (OV_{b}), and an OVPredict with mutex-access metadata optimizations but no bounded CCS tracking (OV_{u}).

We aim to answer the research questions: (1) How effective is OVPredict in predicting order violations? (2) How effective are the optimizations in terms of precision and performance?

3.6.1 Experimental Setup

We compare OVPredict to TSAN [29], a mature widely used data race detector, on both Go and C/C++ programs. For UAF errors, we also compare OVPredict with UFO [7] on several C/C++ concurrent UAF benchmarks. We configured OVPredict and TSAN to report both data races, OV bugs on Go programs, and concurrent UAF bugs on C/C++

Projects	Tests	Races			Order Violations			Total			Overhead		
		TS	$\mathsf{OV_b}$	$\mathsf{OV_u}$	TS	$\mathsf{OV_b}$	$\mathsf{OV_u}$	TS	$\mathsf{OV_b}$	$\mathsf{OV_u}$	TS	$\mathsf{OV_b}$	$\mathsf{OV_u}$
Kubernetes	500	4	9	9	2	4	4	6	13	13	162%	156%	235%
gRPC	14	0	0	0	1	1	1	1	1	1	74%	79%	89%
Prometheus	185	2	4	4	3	3	3	5	7	7	70%	83%	110%
Go-ethereum	509	3	4	4	2	2	2	5	6	6	213%	315%	402%
Syncthing	93	0	1	1	0	0	0	0	1	1	122%	130%	158%
CockroachDB	51	1	1	1	1	2	2	2	3	3	284%	494%	587%
tidb	18	1	1	1	0	0	0	1	1	1	105%	131%	184%
Total	1319	11	20	20	9	12	12	20	32	32	131%	160%	206%

Table 3.2: Go projects results.

programs. All experiments were performed on Docker containers running Ubuntu 18.04 on top of a Google Cloud C2 instance [55] with a 16-hyperthread Intel Xeon CPU and 32GB memory.

Go Benchmarks. Since there are no Go benchmarks for data races and order violations, we applied OVPredict and TSAN to the tests in 7 open-source Go projects from Github. Among all the packages that contain concurrent tests, we selected 69 that use at least one of the annotated libraries as the benchmark. We compare OVPredict and TSAN in terms of plain and annotated race conditions found in five rounds of runs on all 1319 tests from these packages.

3.6.2 OVPredict on Open-source Go Projects

We set a 10-second timeout to each test and counted races with the same pairs of static program locations as one bug. The number of bugs for each tool is accumulated over 10 runs. The results are summarized in Table 3.2.

New OV Bugs Found. OVPredict found 3 new OV bugs on condition variables in Kubernetes, 1 OV bug on condition variables in Cockroach, and 1 OV bug on the context object in Syncthing. We manually verified that all OV bugs are harmful. Specifically, the 4 OV bugs on condition variables can lead to blocked goroutines under specific interleavings. All OV bugs are reported only by OVPredict.

```go
func (s *server) gossipReceiver(...)
{
    s.mu.Lock()
    ...
    if cycler := s.simulationCycler; cycler != nil {
        cycler.Wait()
    }
    s.mu.Unlock()
}

// Gossip embeds a server, so s and g alias here.
func (g *Gossip) SimulationCycle() {
    g.mu.Lock()
    defer g.mu.Unlock()
    if g.simulationCycler != nil {
        g.simulationCycler.Broadcast()
    }
}
```

Figure 3.4: A non-racy COV in CockroachDB.

We have reported all the findings to the developers. All 3 bugs in Kubernetes have been confirmed and fixed. We also checked the commit history of Kubernetes and found the latent periods (*i.e.,* the time between the finding and introducing the bug) of these bugs range from 3 months to 4 years and a half, with a medium value being 3 years. Note that all Go applications we evaluated are mature and actively maintained. Some of the Go projects in Table 3.2, including Kubernetes and CockroachDb, have automated testing pipelines that continuously run stress testing. This shows that the bugs detected by OVPredict are deep and hard-to-find concurrency bugs.

In total, OVPredict found 12 more bugs than TSAN, including 9 more plain races and 3 more annotated race conditions. We manually inspected the reports and confirmed that all bugs reported by TSAN were also reported by OVPredict. Race conditions over condition variables account for 15 of all 20 order violations reported by OVPredict. Such races are not reported by the original TSAN, tend to be highly latent, and the extra delay or leaked goroutines can be hardly noticeable.

Figure 3.4 shows a non-racy COV found in CockroachDB by OVPredict. CockroachDB is a popular open-source distributed SQL database with over 18k Github stars and studied in the survey [4]. The COV, between the broadcast and wait on the condition variable `SimulationCycler`, leads to potential deadlock of the server thread that waits on `cycler`. The purpose of simulation cycles is to make all listening nodes pause and synchronize. However, there is no guarantee that the `gossipReceiver` will always be woken up by the broadcast in `SimulationCycler`, which is repeated only bounded number of times. If the broadcast (line 18) happens before the wait (line 6) on `SimulationCycler`, the broadcast signal will be lost, causing the client to be blocked on wait.

Performance Comparison. Table 3.2 shows the average overhead for all three tools, which is computed as the geometric mean of the additional amount of time relative to the uninstrumented baseline testing runs. OVPredict on average incurs 160% overhead, about 22.1% slower than TSAN. OVPredict has comparable performance with TSAN on gRPC and Syncthing, while performing much worse on the Cockroach and Go-ethereum tests, with a slow down up to 494%.

As discussed in Chapter 2, the additional overhead of OVPredict comes from two main sources. The CCS tracking in OVPredict incurs additional overhead for each memory access in the critical section. OVPredict also keeps both the happens-before clock and the weak-happens-before clock per thread and per synchronization object.

Effectiveness of Bounded CCS Tracking. The bounded CCS tracking optimization, which trades precision for speed, does not make OVPredict report less races or OV bugs compared to an OVPredict without this optimization, as shown in Table 3.2. Furthermore, this optimization significantly improves the performance of OVPredict. OVPredict with this optimization is on average 22.3% faster compared to OVPredict with unbounded CCS tracking and mutex-access metadata optimization.

Program	LOC	#Thr	#Acc	#Lock	#Free
Pbzip2 0.9.4 [56, 7]	1.5K	3 (1)	503	9	17
Apache 2.0.48 [56]	170K	2 (1)	7.1M	5.1K	145K
MySQL 4.0.19 [56]	362K	20 (1)	33.6M	250K	766K
Chromium#841280 [1]	200	39 (4)	405M	363K	27M
Chromium#904714 [57]	106	46 (5)	39M	132K	2.7M
Chromium#944424 [58]	5.5K	41 (4)	432M	551K	36M
Chromium#945370 [59]	31	42 (5)	209M	1.5M	12M

Table 3.3: **UAF Benchmarks.** LOC: lines of code for the non-Chromium programs and for the tests in Chromium bugs. "#Thr": Total number of threads created (with total number of processes created in the parentheses). "#Acc": number of memory accesses. "#Lock": number of mutex lock events. "#Free": number of memory free events.

Benign Race Conditions. However, OVPredict can report high-level race conditions that are benign. We observed several benign races on OV bugs in our evaluation. First, OVPredict does not distinguish benign benign and harmful race conditions. This leads to OVPredict reporting several benign races on condition variables and the context objects. For example, some programs use spinning loops to query if the `err` field of a context object is updated. OVPredict does not comprehend such ad hoc synchronizations currently. We note that by properly annotating library APIs, developers can tune OVPredict to filter out desirable race conditions.

For plain data races, the soundness of predictive analyses employed by OVPredict comes with a few caveats in prior work [21, 20], although in practice, predictive race detection rarely reports false positives. No false positives have been observed in prior work [21, 20, 23]. We inspected a small subset of all the plain races reported and did not find false positives.

3.6.3 Concurrent UAF Detection

We also evaluated OVPredict on a set of benchmarks listed in Table 3.3 for predicting concurrent UAF errors, including: (1) three C/C++ programs containing known UAFs studied in prior works [56, 7] and available from [60], and (2) four concurrent UAF bug reproducers collected from the Chromium issue tracker [61]. The three non-Chromium programs are the only benchmark we can find for concurrent UAF bugs in the literature. The trace statistics of the benchmarks are also reported.

We used the public available instructions to reproduce the concurrent UAF bugs for each benchmark. All non-Chromium tests contain more than one concurrent UAF bug and we ran all three tools 10 times on them. We did not apply UFO to the Chromium tests for two reasons: (1) the offline SMT solving could not finish in one minute for each run, and (2) UFO can be unsound because it ignores reader locks, which are prevalent in Chromium. Each Chromium test contains only one known UAF race and we ran OVPredict and TSAN 1000 times on them to collect the rates for successful detection.

For this evaluation, we used OVPredict to detect concurrent UAF errors only and compared it to a version of TSAN with the same optimization applied. Both tools also employ a compile-time optimization that removes provably redundant instrumentation for memory accesses to the same address within the same basic block, which is similar to the work [62]. We used the same parameters for the window size and the solver timeout in [7].

Existing UAF Benchmark. We first evaluate OVPredict on existing benchmarks used by previous studies. Table 3.4 top shows the results for Pbzip2, Apache and MySQL. All three tools found the same UAF races in Apache. We focus on discussing the results on the two other benchmarks below.

Pbzip2. TSAN and OVPredict respectively found 2 and 3 new UAF races involving concurrent use and destruction of mutexes on Pbzip2, which are missed by UFO. We manually examined the UFO implementation and found that UFO did not process mutex destroy events as free events. We speculate that UFO can also detect these new races if the mutex destroy events are processed.

MySQL. The MySQL benchmark contains one known data race [60] but no previously known UAF races. Both TSAN and OVPredict detected a previously unknown use-free race between a `free_root` call in handle_one_connection and a read from the `fputs` function, while UFO

Program	UFO		TSAN		OVPredict	
	UAF	Online (Offline)	UAF	Time	UAF	Time
Pbzip2	4	1.61 (0.15)	6	0.16	7	0.15
Apache	3	112.3 (1.34)	3	1.79	3	1.90
MySQL	0	671.0 (1.93)	1	2.47	2	2.78

Chromium Bug	TSAN		OVPredict	
	%UAF	Time	%UAF	Time
Chromium#841280	7.31%	0.14	7.38%	0.15
Chromium#904714	100%	0.29	100%	0.39
Chromium#944424	2.0%	0.11	4.4%	0.13
Chromium#945370	13.6%	0.15	14.4%	0.19

Table 3.4: Results for UAF races. The "Time" columns are the average running time in seconds. The "UAF" columns show the number of UAF races detected. Each UAF race is uniquely identified by a pair of static program locations. The "%UAF" columns show the rates of successful detection of the known UAF race. We report both online and offline analysis time for UFO. OVPredict and TSAN only use online analysis. Each non-Chromium test is run 10 times and each Chromium test is run 1000 times.

detected no UAF races. OVPredict also detected a UAF race on a global `COND_thread_count` object that is first accessed in a thread that manages client request and then freed in the main thread.

Historical Chromium UAF. As shown in the bottom of Table 3.4, Both OVPredict and TSAN find all 4 concurrent UAFs using the public bug reproducers. OVPredict has a geometric mean of 32.9% higher detection rates relative to TSAN. In Chromium#904714, both tools find the UAF race in all executions. We inspected the bug reproducer for Chromium#904714 and found that the reproducer is crafted to reliably trigger the UAF in the `SwiftShader` component under AddressSanitizer. This causes the use event to always happen before the free event in all testing runs.

3.6.4 Case Studies: Small Benchmarks

Pbzip2. `Pbzip2` is a parallel file compressor frequently studied in prior work [7, 56] with a number of known use-free races. We use the same input file for all three tools. UFO reported more UAF errors than OVPredict but with less statically distinct free locations, while TSAN reported the fewest number of errors. The free locations reported by OVPredict but missed by UFO are in the main function after `queueDelete` is called (`pbzip.cpp` line 1917-1925), where two mutex objects are destroyed then freed while still being held by another thread.

Apache. `Apache Httpd` contains a known double free error [56] caused by a bogus implementation of reference counting in `mod_mem_cache.c`. In each trial, we concurrently send a number of requests to the server. All three tools can detect this error, despite with different number of erros reported. OVPredict reported the most number of UAF errors, despite sharing the same free locations as UFO. TSAN reported the fewest number of errors in this benchmark.

Mysql. `Mysql` is a benchmark known to contain a data races [56] but it is not yet known if it also contains UAF errors. In each trial, we concurrently fire a number of commands to the mysql client. Both TSAN and OVPredict detected a previously unknown use-free race between a `free_root` call in handle_one_connection and a read from the `fputs` function. OVPredict also detected a predictive UAF on a global `COND_thread_count` object first accessed in a thread that manages client requests then freed in the main thread.

3.6.5 Case Studies: Chromium Bugs

Bug 841280. This bug [1] is showcased in Fig. 3.6, where a global `GCInfo` table is first accessed by the main thread then accessed by the shared worker thread. The access to the table from the main thread is not protected by locks. Both TSAN and OVPredict can detect this use-free race with a similar but low rate, while UFO only detected this bug in 6 out of the ten trials we conducted.

Bug 904714. Figure 3.5a shows a code snippet of the bug [57], which is in the `SwiftShader` component of Chromium and is caused by the `sw::Query` objects being freed in `glDeleteQueries` while still in use by the renderer. When the renderer shuts down, it dereferences the query object, leading to an exploitable UAF. Both TSAN and OVPredict can stably report the

```
// use location
void Renderer::finishRendering(...) {
...
for(auto &query : *(draw.queries)) {
    switch(query->type) {
    ...
...
}

// free location,
void QueryManager::RemoveQuery(...) {
...
query->Destroy(...);
...
}
```

(a) Bug 904714.
UAF in `sw::Renderer::finishRendering`.

```
// used by main thread
TaskRunner CreateTaskRunner(...) {
// check if "impl_" is reset
if (!impl_)
    return ...;
// "impl_" is not yet reset
return impl_->Create...;
}

// freed by SharedWorker thread
void DidRunTask() {
...
// Delete all task queues
queues_to_delete.clear();
}
```

(b) Bug 944424.
UAF in `TaskQueueImpl::CreateTaskRunner`

```
// use location
void DB::DeleteDatabase(...) {
...
AppendRequest(this);
...
}

// free location
void DB::RequestComplete(...) {
...
delete this;
}
```

(c) Bug:945370. UAF in IndexedDB.

Figure 3.5: UAF case studies on Chromium.

```
// Main thread. No locks held.
T GCInfo(int index) {
  T* table = g_gc_info_table;
  T info = table[index]; // use
  return info;
}

// SharedWorker thread
void EnsureGCInfoIndex() {
  lock(m);
  gc_info_index++;
  if (gc_info_index >= gc_info_table_size) {
    // Capacity reached. Resize g_gc_info_table.
    free(g_gc_info_table);
    ...
  }
  unlock(m);
}
```

Figure 3.6: A UAF race in Chromium Blink garbage collector [1].

UAF error as a use-free race in all trials, while this UAF is detected 6 times out of 10 trials for UFO.

Bug 944424. This bug [58], listed at Fig. 3.5b, is caused by a race condition between the `TaskQueue` disposal when the worker thread terminates and the `GetTaskRunner` function called from the main thread. Using the reproducer provided by the bug reporter, OVPredict can trigger the error with a higher rate than TSAN, despite the rate being extremely low. We speculate that such a low rate is caused by the narrow race window in the interaction between the main thread and the worker thread.

Bug 945370. This UAF [59], illustrated in Fig. 3.5c, is in the `IndexedDB` module, caused by a queued active request being destroyed in the `RequestComplete` function called upon the closing of the database. The request is later dereferenced in `DeleteDatabase`. The reproducer triggers the bug by repeatedly sending a sequence of open and delete requests to the database. OVPredict detected this bug with a higher rate than TSAN, while this bug did not manifest in our trials of UFO. Nonetheless, the predictive analysis employed by OVPredict increased the likelihood of detecting the bug.

3.7 Conclusion

We presented OVPredict, an efficient partial-order-based predictive analysis for OV bugs that scales to long running traces for Go and C/C++ programs. OVPredict extends the WDC partial order to target order violation bugs as well as data races. OVPredict uses two new optimizations for tracking CCS ordering, and exploits shadow memory for time and space efficiency. Compared with SmartTrack, the OVPredict optimizations have simpler logic and are more understandable. OVPredict uncovers 5 new OV bugs in popular open-source Go projects, including Kubernetes and CockroachDB, and detects more UAF races than TSAN in C/C++ benchmarks, with a comparable performance cost.

4. NCMC: EXPLOITING SEMANTIC COMMUTATIVITY IN STATELESS MODEL CHECKING

In this chapter we introduce a stateless model checking algorithm NCMC that systematically enumerates all non-redundant interleavings characterized by dependency at the library interface level.

4.1 Introduction

Consider the following program where N threads concurrently add distinct elements to a *set* object s implemented with a linked list. Suppose that the *add*() operation on s is atomic.

$$\begin{gathered} s.add(1); \;\Big\|\; \dots \;\Big\|\; s.add(N); \\ \mathbf{assert}(s.size() = N); \end{gathered} \tag{N-SET-ADD}$$

At the semantics level, the execution order of these *add*() invocations does not matter; all orders yield the same return values for each invocation and produce the same set (albeit not the same linked list). Therefore, exploring only one execution suffices to check the assertion at the end. However, these *add*() invocations do not commute at the instruction level: there exists memory dependence between reads and writes (a.k.a. *reads-from* relation) within these invocations, and their different orders produce different linked lists. Any DPOR algorithms that exploit independence at the instruction level would therefore conservatively conclude that these invocations do not commute. As such, they explore $N!$ possible executions to cover the full state space. These executions may differ in the concrete states of the set s. However, these differences are not observable by the client of the set data structure. Considering all of them is therefore unnecessary.

We introduce the notion of *semantic commutativity equivalence* (SC-equivalence), which is coarser than the ones characterized by partial orders over events at the instruction level. Intuitively, two executions are SC-equivalent if one can be obtained from the other by swapping adjacent commutative operations. Based on this notion, we present NCMC (*non-commutativity model checking*), a new SMC algorithm that is able to effectively exploit semantic commutativity. Our approach uses a commutativity specification[1] on concurrent objects to identify commutative operations in each exploration of the program, and avoids exploring executions that do not cover a new abstract state. One key difference from previous approaches is that our approach covers all *abstract states* rather than *concrete states* of the program. This weaker completeness property, called *semantic completeness*, enables our algorithm to achieve significant reduction in the number of explored executions, while still ensuring that all behaviors of the program at the semantic level are covered.

Our contributions can be summarized as follows:

1. We give an intuitive account of NCMC through a series of examples, and show how NCMC prunes SC-equivalent executions while remaining semantically complete (§4.2).

2. We present the formal model that underpins NCMC (§4.3). There, we also describe different logic fragments that can be used in commutativity specifications.

3. We present NCMC in detail, and show that it is sound (produces no false positives) and semantically complete (explore all possible behaviors at the semantic level) with respect to a sound (albeit not necessarily precise) commutativity specification (§4.4).

4. We implement NCMC atop the JMCR model checker [46] and evaluate it against two state-of-the-art model checkers on two collections of benchmarks (§4.5). Our evaluation shows that NCMC is significantly faster or comparable than JMCR and Yogar-CBMC on several SV-COMP benchmarks with commutative operations, and achieves exponential reductions for a number of concurrent data structure implementations under various workloads. We also demonstrate how the precision of commutativity specifications can affect the performance of NCMC under different workloads.

[1]We assume the commutativity specifications are given by the user. Automatic generation of commutativity specifications has also been studied in the literature [50, 51].

4.2 Overview

4.2.1 Extending Plain Executions with Semantics

Traditional SMC algorithms typically employ DPOR to reduce the number of interleavings that need to be explored. Given a suitable notion of *instruction independence*, DPOR techniques partition all possible interleavings into equivalent classes and explore only one from each class. Each equivalence class of interleavings has the same partial order, which can be naturally represented by a *plain execution graph* or PEG [48, 63]. Nodes in a PEG represent events that correspond to instructions, *e.g.,* a read or a write, and edges represent orderings between nodes. At the level of *instructions*, PEGs can precisely capture sundry orders among *primitive events* (*e.g.,* reads and writes). By enumerating all consistent PEGs of a program, the concrete state space of the program can be efficiently covered. At the level of *semantics*, however, two distinct PEGs may be equivalent with respect to the abstract states of a data structure.

Consider the following program in which N threads concurrently invoke *inc()* operations on an atomic counter c:

$$c.inc(); \;\Big\|\; ... \;\Big\|\; c.inc(); \tag{N-COUNTER-INC}$$

Let us assume the $inc()$ method is implemented as $lock(l); a := x; x := a + 1; unlock(l)$, where x is the shared variable that keeps track of the counter state, and a is a local variable. Since the reads and writes in two consecutive $inc()$ operations are dependent, existing DPOR algorithms that exploit instruction-level independence will explore $N!$ executions to cover all states of the program.

Consider $N = 2$, DPOR will yield two PEGs as depicted in Fig. 4.1, in which the solid black edges denote the *program order* (`PO`), and the green dashed edges denote the *reads-from* relations (`RF`). The two executions Ⓐ and Ⓑ are different in that they have distinct `RF`

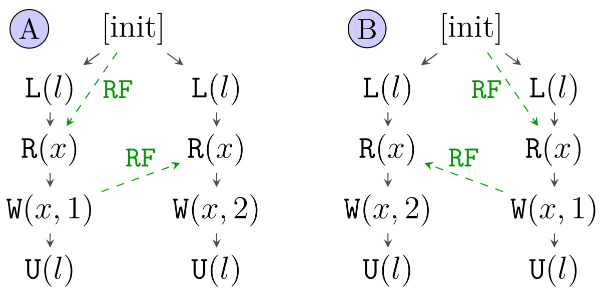


Figure 4.1: Two plain execution graph (PEG)s of N-COUNTER-INC with $N = 2$.

edges. Specifically, assume without loss of generality that the DPOR algorithm first obtains execution Ⓐ, where the read in the second thread (the thread on the right) reads from the write in the first thread (the thread on the left). The algorithm subsequently makes the read in the second thread to read from the initial value, yielding execution Ⓑ.

Redundancy at the Semantics Level. The limitation of traditional DPOR algorithms is that they are oblivious to the program semantics. Indeed, exploring both executions Ⓐ and Ⓑ are necessary to cover the full state space of the program. However, they are redundant at the semantics level: the two *inc*() method invocations are commutative. More specifically, the order of which *inc*() executes first does not matter; at the end of the execution both orders yield the same state for counter *c*. Although their orders affect the values read by the read events within each *inc*() invocation, such differences only exist at the concrete implementation level for the counter object, and are not observable to the clients of the counter object.

Semantic Execution Graph. To exploit semantic commutativity between method invocations of concurrent data structures, we introduce the notion of *semantic execution graph (SEG)*s, an extension to PEGs. SEGs extend PEGs in two ways:

- First, in a SEG, events that happen within the same invocation are replaced with an *invocation node*. For example, as illustrated in Fig. 4.2, the events within the two dashed rectangles in Ⓐ are replaced with their corresponding invocations in Ⓑ. This allows us to consider events at the granularity of both instruction and invocation levels.

- Second, at the granularity of invocations, we are interested in their dependence at the semantics level, and not in the concrete implementation level. That is, we do not consider the RF relation between reads and writes within these invocations. Instead, we introduce the NC relation (*noncommutative causal order*) to capture the dependence between noncommutative invocations on the same object. This enables us to keep semantically commutative but implementation-wise dependent invocations unordered.

The NC Relation in Semantic Executions. Intuitively, the NC relation captures dependence between pairs of noncommutative invocations on the same object. Every pair of noncommutative invocations in one semantic execution is NC-ordered. On the other hand, two adjacent invocations that are unordered by NC in a linear extension of one execution can be swapped without making the execution inconsistent. In addition, we make the initialization event for each object NC-ordered before all invocations on it. For example, the two *inc*() invocations in the semantic execution Ⓑ of Fig. 4.2 are not ordered by NC, but they are both NC-ordered from the initialization event. We will present NC formally in §4.3.

As we will see later, the semantic equivalence underpinned by the NC relation is coarser than the equivalence relations obtained by instruction independence, which is employed by existing SMC algorithms. Underpinned by this new notion of semantic equivalence, NCMC prunes semantically redundant executions while still ensuring all abstract states of the program are covered. For instance, in the N-COUNTER-INC example with $N = 2$, suppose NCMC arrives at the execution Ⓐ in Fig. 4.2 in the first exploration. Since the two *inc*() invocations are not ordered by NC, NCMC will not consider the new reads-from (RF) option for the read in the second thread, and thus avoids exploring Ⓑ in Fig. 4.1. In this case, NCMC explores only one execution and terminates. Similarly, in the general case with N threads, NCMC explores *only one* execution rather than $N!$ executions.

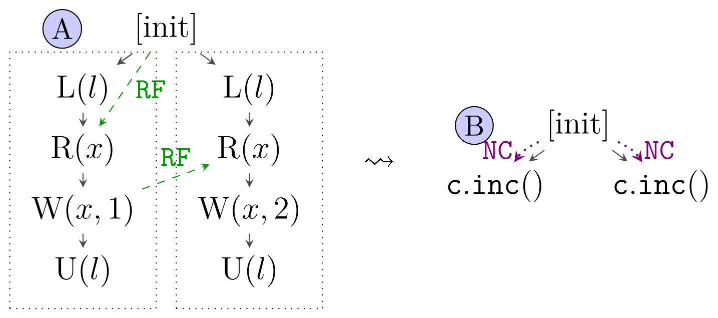


Figure 4.2: A plain execution and its semantic execution of N-counter-inc with $N = 2$.

4.2.2 Online Construction of Semantic Execution Graphs

The online construction of semantic execution graphs (SEGs) is a significant part of our contribution. Specifically, NCMC builds a SEG on top of a plain execution incrementally during each exploration. NCMC detects when an invocation ends by instrumenting the source program, and adds the invocation node to the SEG at the end of the invocation.

The key part in constructing the SEGs is inferring the NC order, which requires identifying noncommutative pairs of invocations. Consider the following program where two threads concurrently invoke operations on a *set* data structure s:

$$s.add(1); \quad \Bigg\| \quad \begin{array}{l} s.add(2); \\ s.contains(1); \end{array} \qquad \text{(Set-2add+cont)}$$

The *add*() and *contains*() methods are provided by the set interface. The return values of $s.contains(1)$ depend on whether it executes before $s.add(1)$. It returns `false` if it executes before $s.add(1)$ and `true` otherwise. As such, we say that the two invocations are *non-cummutative*. To decide whether two invocations commute, we resort to a commutativity specification, explained next.

Commutativity Specifications. A commutativity specification captures commutativity conditions between method invocations in a declarative way with logical formulas. Specifically, for every pair of methods of a given data structure, the specification describes when

	$s.add(v_2)\backslash r_2$	$s.contains(v_2)\backslash r_2$
$s.add(v_1)\backslash r_1$	$v_1 \neq v_2 \vee (r_1 = \bot \wedge r_2 = \bot)$	$v_1 \neq v_2 \vee r_1 = \bot$
$s.contains(v_1)\backslash r_1$	$v_1 \neq v_2 \vee r_2 = \bot$	$\top$

Table 4.1: A commutativity specification of the *set* data structure.

two methods commute. For example, the commutativity specification of a set data structure is shown in Table 4.1. For brevity, we only list the commutativity conditions between *add*() and *contains*() methods, and we use $\top$ to denote `true` and use $\bot$ to denote `false`. Each method invocation is denoted in the form $o.m(\vec{x})\backslash\vec{r}$ where o is the object, m is the method, and $\vec{x}$ and $\vec{r}$ are respectively arguments and return values of the invocation.

In this example, $s.add(1)$ and $s.add(2)$ commute regardless of their return values, while $s.add(2)$ and $s.contains(2)$ commute only when $s.add(2)$ returns `false`, meaning the state of s is not modified.

On-the-fly Construction of SEGs. During each execution of the program, NCMC adds invocation nodes (along with primitive event nodes) to the SEG one at a time and infers the `NC` relation on the fly. Consider the Set-2add+cont example. Starting from an initial graph (containing initialization events only), NCMC first adds the $s.add(1)\backslash\top$ node from the first thread to the graph, arriving at execution ① of Fig. 4.3. Next, it adds $s.add(2)\backslash\top$ from the second thread. To obtain the dependence between previous invocations and the $s.add(2)\backslash\top$ node, NCMC evaluates the commutativity condition between $s.add(2)\backslash\top$ and all previous invocations on s. Here, there is only one invocation, $s.add(1)\backslash\top$, before $s.add(2)\backslash\top$. Since the commutativity condition is evaluated to $\top$ for $s.add(2)\backslash\top$ and $s.add(1)\backslash\top$, they are commutative and are not `NC`-ordered. Instead, NCMC adds an `NC` edge from the [init] node to $s.add(2)\backslash\top$, arriving at ② in Fig. 4.3. Finally, NCMC adds $s.contains(1)\backslash\top$ to the graph, which does not commute with the previous invocation $s.add(1)\backslash\top$ according to the commutativity condition. Consequently, an `NC` edge is added between them, arriving at ③ in Fig. 4.3.

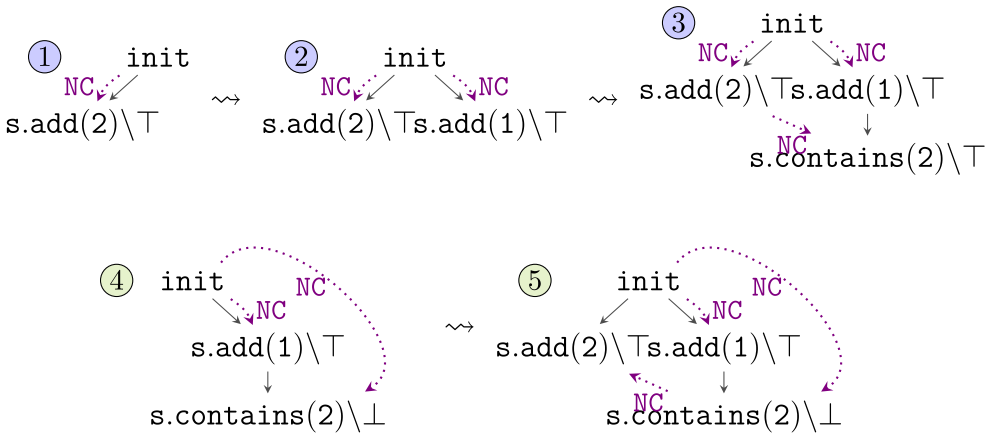


Figure 4.3: Key steps for constructing the semantic execution graph (SEG).

NCMC proceeds by exploring a new execution for SET-2ADD+CONT. The key steps for constructing its SEG, depicted by ④ and ⑤ in Fig. 4.3, are similar to those of the first execution.

4.2.3 Handling Events at Both Instruction Level and Semantics Level

NCMC is capable of correctly handling events at both the instruction level and the semantics level. Specifically, NCMC partitions the reads in a plain execution into two classes, *scoped reads* and *unscoped reads*, which contain the read events within or without an invocation respectively. NCMC only exploits semantic commutativity for scoped reads, and handles unscoped reads as its underlying SMC algorithm does. Handling scoped reads is subtle in the presence of conflicting events between a pair of invocations. Let us illustrate this via the following program:

$$\left.\begin{array}{l} c.inc(); \\ a := x; \\ \textbf{if } a = 0 \textbf{ then } x := 2; \end{array}\right\| \begin{array}{l} x = 1; \\ c.inc(); \end{array} \qquad (\text{COUNTER-2INC-CF})$$

As before, starting from an initial SEG, NCMC incrementally adds all events from the first thread and then all events from the second thread, arriving at execution Ⓐ in Fig. 4.4.

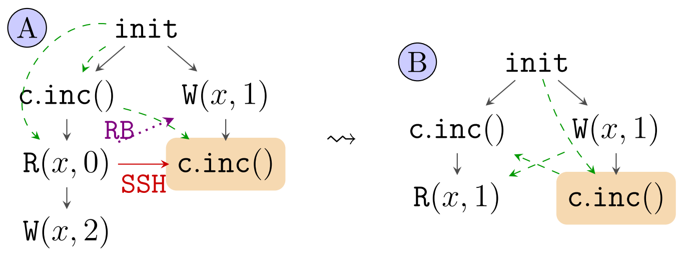


Figure 4.4: Key steps in exploring COUNTER-2INC-CF, with invocation graphs shown on the right.

Here, we also add a RF edge between a pair of invocations if they contain a pair of read and write linked by RF. Similar to N-COUNTER-INC, the two *inc*() invocations commute. In addition, the node R$(x, 0)$ reads from the initial write, implying a *reads-before* order before the write W$(x, 1)$. In other words, since R$(x, 0)$ reads from the initial write, it must be ordered before the write node W$(x, 1)$ to ensure coherence.

Now NCMC partitions reads in the execution Ⓐ into scoped reads and unscoped reads. The former consists of the two reads within the two *c.inc*() invocations, while the latter consists of the R$(x, 0)$ in the first thread. For the unscoped read R$(x, 0)$, NCMC explores its reads-from option, W$(x, 1)$ from the second thread as its underlying SMC algorithm does. Different from N-COUNTER-INC, however, for the scoped read in *c.inc*() from the second thread, NCMC needs to explore the reads-from option, *i.e.,* the write in `init`. This new exploration, depicted as Ⓑ, is not redundant because the read $a := x$ in the first thread will then read from the write $x = 1$ in the second thread, yielding a different semantic execution graph.

Let us demonstrate when NCMC deems a reads-from option necessary to explore in COUNTER-2INC-CF. In execution Ⓐ, the union of RB and PO imposes a *semantic-happens-before* (SSH) between the R$(x, 0)$ node and the *c.inc*() from the second thread. NCMC therefore concludes that this reads-from option will yield a new execution where the SSH edge is not preserved. That is, by making the read in the highlighted *c.inc*() read from `init`, the read $a := x$ will be ordered after the highlighted *c.inc*() node (as is shown in Ⓑ). NCMC ensures that the execution Ⓑ, which covers a new semantic state of the program, is explored.

Intuitively, the SSH order captures the causality at both the semantic level and the instruction level, and is induced by NC, PO, RB and RF, to be defined at §4.3. When considering a RF option for a scoped read, NCMC checks if all SSH edges associated with the invocation that contains the read are preserved in the new exploration. If it is the case, the RF option will lead to a *SC-equivalent execution* and can be safely pruned. Otherwise, the reads-from option may or may not lead to a new semantic state of the program, *i.e.,* not a SC-equivalent one. As we will demonstrate in §4.4, in such cases, NCMC will use additional constraints to ensure only semantically new executions are explored. In both cases, NCMC ensures that all abstract states of the program are eventually covered while no executions that are SC-equivalent to the current one are explored.

4.3 The Formal Model

This section presents the formal model underpinning NCMC. We first adopt notation and terminology from the literature of declarative (a.k.a. axiomatic) memory models to define events and PEGs [48, 53, 63] (Section 4.3.1). We then introduce the notions of invocations (Section 4.3.2), commutativity (Section 4.3.3), commutativity specifications (Section 4.3.4), and the semantic-happens-before relation SSH (Section 4.3.5) used by our algorithm.

Notation. Given a relation $\mathtt{r}$, we write $\mathtt{r}^+$ for the transitive closure of $\mathtt{r}$. We write $\mathtt{r}^{-1}$ for the inverse of $\mathtt{r}$; $r|_A$ for $r \cap (A \times A)$; $[A]$ for the identity relation in $A : \{\langle a, a\rangle \mid a \in A\}$. Given relations $\mathtt{r}_1$ and $\mathtt{r}_2$, we write $\mathtt{r}_1 ; \mathtt{r}_2$ for $\{(a, b) \mid \exists c.(a, c) \in \mathtt{r}_1 \wedge (c, b) \in \mathtt{r}_2\}$, *i.e.,* their relational composition. When $\mathtt{r}$ is a strict partial order, we write $\mathtt{r}|_{\mathtt{imm}}$ for $\mathtt{r} \setminus \mathtt{r} ; \mathtt{r}$, *i.e.,* the *immediate* edges in $\mathtt{r}$.

4.3.1 Labels, Events and Plain Executions

We represent the traces of a program as a set of execution graphs, where the graph nodes denote *events*, and graph edges capture different kinds of relations over these events. Each event, denoted as a tuple $\langle i, l\rangle$, corresponds to the execution of an instruction, where $i \in$

$\mathsf{Tid} \uplus \{0\}$ is a *thread identifier* (0 for initialization events) with $\mathsf{Tid} \subseteq \mathbb{N}$ and $l \in \mathsf{Lab} \uplus \{\mathtt{error}\}$ is an event *label*.

A label maybe either:

1. the `error` label, denoting assertion violations; or

2. an *instruction label* $l \in \mathsf{Lab}$. For instance, the write event is associated with the label $\mathtt{W}(x, 1)$. We also assume a set of memory locations, Loc, ranged over by x, y, z. We further assume a set of *read labels*, $\mathsf{RLab} \subseteq \mathsf{Lab}$, a set of*write labels*, $\mathsf{WLab} \subseteq \mathsf{Lab}$, a set of *lock labels*, $\mathsf{LLab} \subseteq \mathsf{Lab}$, a set of *unlock labels*, $\mathsf{ULab} \subseteq \mathsf{Lab}$, a set of *fork labels*, $\mathsf{FLab} \subseteq \mathsf{Lab}$, and a set of *join labels*, $\mathsf{JLab} \subseteq \mathsf{Lab}$, associated with *read*, *write*, *lock*, *unlock*, *fork* and *join* instructions, respectively. For instance, the label $\mathtt{W}(x, 1)$ is a write label.

Definition 4.3.1 (Events)**.** A (primitive) *event* $e \in \mathsf{Event}$ is a tuple $\langle i, l \rangle$ where $i \in \mathsf{Tid}$ is a thread identifier, and $l \in \mathsf{Lab} \uplus \{\mathtt{error}\}$ is a label.

We define the functions $\mathsf{tid}()$ and lab to project the thread identifier and the label of a event, respectively; the functions $\mathsf{loc}()$, $\mathsf{val}_{\mathtt{r}}$ and $\mathsf{val}_{\mathtt{w}}$ project the location, the read or the written value of a label respectively, where applicable. For instance, $\mathsf{loc}(()l) = x$ and $\mathsf{val}_r(l) = 1$ for $l = \mathtt{R}(x, 1)$. We lift the functions $\mathsf{loc}()$, $\mathsf{val}_{\mathtt{r}}$ and $\mathsf{val}_{\mathtt{w}}$ to events as well, *e.g.,* $\mathsf{loc}(()e) = \mathsf{loc}(()\mathsf{lab}(e))$ for an event e. The functions $\mathsf{tid}()\mathtt{F}$ and $\mathsf{tid}()\mathtt{J}$ project the thread identifier of a fork or join label. For instance, $\mathsf{tid}()\mathtt{F}(l) = 2$ for $l = \mathtt{F}(2)$. We define the *read events* as $\mathtt{R} \triangleq \{e \in \mathsf{Event} \mid \mathsf{lab}(e) \in \mathsf{RLab}\}$; the write ($\mathtt{W}$), lock ($\mathtt{L}$), unlock ($\mathtt{U}$), fork ($\mathtt{F}$), join ($\mathtt{J}$) events are defined analogously. Given a set of events E, we write E_x for $\{e \in E \mid \mathsf{loc}(()e) = x\}$. We define *initialization events* as $\mathsf{Event}_0 \triangleq \{e \in \mathtt{W} \mid \mathsf{tid}(()e) = 0\}$. We write Val for the set of all possible program arguments and return values.

Definition 4.3.2 (Plain Execution Graphs)**.** A *plain execution graph* (PEG) is a tuple $G = \langle E, \mathtt{PO}, \mathtt{RF} \rangle$, where $E \subseteq \mathsf{Event}$ is a set of *events*, $\mathtt{PO}$ is the *program order*, and $\mathtt{RF} : E \cap \mathtt{R} \to E \cap \mathtt{W}$

is the *reads-from* function. We write $G.\mathtt{E}$, $G.\mathtt{PO}$ and $G.\mathtt{RF}$ for its components, and write $G.\mathtt{R}$ for $G.\mathtt{E}\cap\mathtt{R}$; similarly for $G.\mathtt{W}$, $G.\mathtt{L}$, $G.\mathtt{U}$, $G.\mathtt{F}$, $G.\mathtt{J}$ and $G.\mathtt{I}$. We also write $G.\mathtt{E}_0$ for $G.\mathtt{E}\cap\mathsf{Event}_0$.

Given a PEG G, we define the *must-happen-before* order ($G.\mathsf{MustHappenBefore}$), the *write order* ($G.\mathtt{WO}$), the *reads-before* order ($G.\mathtt{RB}$), and the *happens-before* order ($G.\mathtt{HB}$) on G.

The $\mathsf{MustHappenBefore}$ Relation. The $\mathsf{MustHappenBefore}$ relation is defined to be the union of $\mathtt{PO}$ and the partial order imposed by fork and join events. In particular, (1) a fork event e must happen before all events in the thread forked by e; (2) a join event e must happen after all events in the thread to be joined by e. Formally,

$$G.\mathsf{MustHappenBefore} \triangleq G.\mathtt{PO}\cup\{\langle\langle i_1, l_1\rangle, \langle i_2, l_2\rangle\rangle \mid (l_1 \in \mathtt{FLab}\wedge\mathtt{tid}_{\mathtt{F}}(l_1) = i_2)\vee(l_2 \in \mathtt{JLab}\wedge\mathtt{tid}_{\mathtt{J}}(l_2) = i_1)\}$$

The $\mathtt{WO}$ Relation. The $\mathtt{WO}$ relation denotes the *write order* between two writes on the same location. We define $\mathtt{WO}$ to be the disjoint union of the write order for each variable, *i.e.*, $\mathtt{WO} \triangleq \bigcup_{x\in Loc} \mathtt{WO}_x$. Given a variable x, its write order $\mathtt{WO}_x$ is defined to be the total order of the write operations performed on it.

The $\mathtt{RB}$ Relation. Intuitively, $\mathtt{RB}$ relates each read r to the writes that are $\mathtt{WO}$-after the write r reads from. We define $G.\mathtt{RB}$ as: $G.\mathtt{RB} \triangleq G.\mathtt{RF}^{-1}; G.\mathtt{WO}$.

The $\mathtt{HB}$ Relation. The *happens-before* relation $\mathtt{HB}$ on G is the smallest transitive relation that contains $\mathtt{PO}$, $\mathtt{WO}$ and $\mathtt{RB}$: $G.\mathtt{HB} \triangleq (G.\mathtt{PO} \cup G.\mathtt{WO} \cup G.\mathtt{RB})+$.

4.3.2 Invocations

We represent a *data structure interface* $\mathtt{DS}$ as a set of methods $\mathtt{M} \subseteq \mathsf{Method}$. We write Method^L for the set of *linearizable* methods, and write $\mathtt{DS}^L$ to restrict the data structure interface to linearizable methods, *i.e.*, $\mathtt{DS}^L \triangleq \{m \mid m \in \mathtt{M} \cap \mathsf{Method}^L\}$. Intuitively, linearizability provides the illusion that any operation performed on a concurrent data structure takes effect instantaneously at some point between its invocation and its response [64]. Note that this means we only need to consider interleavings of method invocations on these objects at the granularity of invocations, and there is a total ordering among all invocations for one object. In other words, we only consider linearizable invocations. For non-linearizable invocations, we consider their underlying events (*e.g.,* reads and writes) instead.

As discussed in §4.2, an invocation consists of a sequence of events that happen during the method invocation. As the commutativity conditions may reference the abstract state just before a method is invoked, we are also interested in the method, arguments and return value of an invocation, which are associated with the label of an invocation. An invocation label $l \in ILab$ is of the form $\sigma_o.m(v_1, v_2, \ldots, v)$, denoting an invocation of method m with arguments $v_1, v_2, \ldots, v$ on object $o \in \mathsf{Obj}$ with state σ_o, where $m \in \mathsf{Method}$ is a method name, $v_i \in \mathsf{Val}$ is an argument and $v \in \mathsf{Val}$ is the return value of the invocation. The functions obj, σ, met, $\mathsf{val}_{\mathsf{arg}}$ and $\mathsf{val}_{\mathsf{ret}}$ respectively project the object, the method, the abstract state, the arguments and the return value of an invocation label. For instance, $\mathsf{obj}(l) = s$, $\sigma(l) = \sigma_s$, $\mathsf{met}(l) = \mathsf{add}$, $\mathsf{val}_{\mathsf{arg}}(l) = [1]$ and $\mathsf{val}_{\mathsf{ret}}(l) = \texttt{false}$, for $l = \sigma_s.add(1, \texttt{false})$.

Definition 4.3.3 (Invocations)**.** An *invocation* $v \in \mathsf{Inv}$ is a tuple $\langle \mathtt{E}, l \rangle$ where $\mathtt{E}$ is a set of events and $l \in \mathsf{ILab}$ denotes the *invocation label* satisfying the condition $\mathsf{met}(l) \in \mathsf{Method}^L$.

We typically use u, v to range over invocations. Given an invocation v, we write $\mathtt{evt}(v)$ to project its set of events E. Given a method $m \in \mathsf{Method}^L$, we write Inv_m for the set of invocations on method m. Given a set of methods $M \subseteq \mathsf{Method}^L$, we write Inv_M for $\bigcup_{m \in M} \mathsf{Inv}_m$. We also define two helper functions, rep and top:

- The function rep chooses some event a from $\mathtt{evt}(v)$ as the *representative* of v. For concreteness, we let rep return first event in $\mathtt{evt}(v)$. We also define rep as the identity function on events, *i.e.,* $\mathsf{rep}(a) = a$. Given a sequence of invocations $I \in \mathsf{Inv}$, we write $u \prec_{\mathtt{I}} v$ if u precedes v in I. We define initialization events for objects as $\mathsf{Inv}_0 \triangleq \{v \in \mathsf{Inv} \mid v.\mathtt{E} \subseteq \mathsf{Event}_0\}$.

- The partial function $\mathsf{top} : \mathsf{Event} \rightharpoonup \mathsf{Inv}$ maps an event to its corresponding invocation, *i.e.,* $\mathsf{top}(e) = v$ if $e \in v.\mathtt{E}$ and $\mathsf{top}(e) = \perp$ otherwise.

Given the top function and a PEG G, We say that an event a is *scoped* if $\mathsf{top}(a) \neq \bot$, *i.e.,* a is not within any invocation; otherwise a is *unscoped*. We write Event^u to restrict a set of events E to unscoped events only, and write Event^s for $\mathsf{Event} \setminus \mathsf{Event}^u$. We use G^u to restrict a PEG G to its unscoped events, *i.e.,* given $G = \{E, \mathtt{PO}, \mathtt{RF}\}$, $G^u = \{E^u, \mathtt{PO}|_{E^u}, \mathtt{RF}|_{E^u}\}$.

4.3.3 Commutativity via Effects of Invocations

Abstract States. We are interested in the *abstract states* of the objects as described by its specification and not in the actual implementation details of the object. We write σ_o for the abstract state of an object o. For example, σ_s denotes the set of elements present in the set s. We assume that object methods are given by specifying the effects on their abstract states. For example, Table 2.1 describes the method effects of a set object. The shared state of a program with respect to a set of objects $O \subseteq \mathsf{Obj}$, S_O, is defined as $S_O \triangleq \biguplus_{o \in O} \sigma_o$. We write S for the set of all shared states.

Effects of Invocations. The effect on the shared state of an invocation v is given by a partial map $(\!|v|\!) \in S \rightharpoonup S$. For example, for a set object s, the map $(\!|s.contains(v)\backslash\mathtt{true}|\!)$ should be the identity on all states in which the set s contains v and undefined otherwise. We say that an invocation v is *enabled* in the shared state σ if the effect of v is defined on σ. For example, the map $(\!|s.contains(v)\backslash\mathtt{true}|\!)$ is enabled on all states in which the set s contains v.

Definition 4.3.4 (Commutativity)**.** Two invocations $u, v \in \mathsf{Inv}$ *commute*, denoted by $u \bowtie v$, iff $(\!|u|\!) \circ (\!|v|\!) = (\!|v|\!) \circ (\!|u|\!)$.

We say that two invocations commute when independent of their application order, their composed effects are the same. For example, following Table 2.1, two add invocations commute when they add different elements, as they modify disjoint parts of the object state σ_s. We assume that actions of different objects always commute, *i.e.,* invocations of one object do not interfere with the state of another object. We write $(\!|v|\!)_\sigma$ for the abstract state after v is invoked on σ.

$$
\begin{array}{lcll}
S & := & \sigma_1 \mid \sigma_2 & \text{Abstract states} \\
V & := & v_1 \mid v_1 \mid r_1 \mid r_2 \mid \mathbb{Z} \mid \mathbb{B} & \text{Arguments, return values, constants} \\
F & := & f(S, V, V, \dots) & (S \times V \times V \times \dots) \to \mathbb{Z} \cup \mathbb{B} \\
O & := & + \mid - \mid * \mid / \mid < \mid > \mid \wedge \mid \vee & \text{Arithmetic, boolean connectives and equity} \\
P & := & V \mid F & \text{Primitive formula} \\
C & := & P\ O\ P \mid (C) \mid \neg C\ \mid C\ O\ C & \text{Formula}
\end{array}
$$

Figure 4.5: The logic L_1 to express commutativity conditions.

	$put(k_2, v_2)\backslash p_2$	$get(k_2)\backslash v_2$	$size()\backslash r_2$
$put(k_1, v_1)\backslash p_1$	$k_1 \neq k_2 \vee (v_1 = p_1 \wedge v_2 = p_2)$	$k_1 \neq k_2 \vee (v_1 = p_1)$	$(v_1 \neq nil \wedge p_1 \neq nil)$
$get(k_1)\backslash v_1$	$k_1 \neq k_2 \vee (v_2 = p_2)$	$\top$	$\top$
$size()\backslash r_1$	$(v_2 \neq nil \wedge p_2 \neq nil)$	$\top$	$\top$

Table 4.2: A commutativity specification of the *hashtable* data structure, expressible in the L_2 logic.

Note that our definition of commutativity is equivalent to the following observation. At a given state σ, if the preconditions of the two method invocations are satisfied (*i.e.,* enabled) in the first execution order, then (1) the preconditions of the invocations are satisfied in the second execution order; (2) the invocations return the same values in both execution order, and (3) the abstract final state are the same in both execution orders.

We write $u \ltimes v$ if u and v do not commute, *i.e.,* different application order of u and v yields different shared states of the program. For example, let $v = s.add(1)\backslash\texttt{true}$ and $u = s.contains(1)\backslash\texttt{false}$, then $v \ltimes u$ because $(\!|v|\!) \circ (\!|u|\!) \neq (\!|u|\!) \circ (\!|v|\!)$. In particular, let σ be the state of the set s such that $1 \notin \sigma$, then $(\!|v|\!) \circ (\!|u|\!)$ is undefined but $(\!|u|\!) \circ (\!|1|\!) = \sigma \uplus \{1\}$.

4.3.4 Commutativity Specifications

We next give a formal definition for commutativity specifications containing logical formulas that represent commutativity conditions for each pair of methods in an object. The commutativity conditions are used by NCMC to identify the `NC` relations in a semantic execution.

Definition 4.3.5 (Commutativity Conditions)**.** Given a suitable logic, a logical commutativity specifications for a pair of methods $m_1, m_2 \in M$ of the same object is given by a logical formula $\psi^{m_1}_{m_2}$ with its free variables collected into the list $(\vec{x_1}; \vec{x_2})$ so that $\vec{x_i}$ match the abstract states, return values and arguments of $v_i \in \mathsf{Inv}_{m_i}$.

Commutativity Conditions. We capture commutativity between method invocations with logical formulas [65, 66]. Specifically, a (sound) logical commutativity condition is a predicate ψ over pairs of invocations $u, v \in \mathsf{Inv}$ such that $\psi(u, v)$ implies that u and v commute. The logic L_1 of these formulas is given in Fig. 4.5. The vocabulary of the logic includes the arguments and return values of method invocations. The specification in L_1 can also use arbitrary functions over the arguments as well as the abstract states in which the methods are invoked. The logic allows boolean connectives, equality and arithmetic, but no quantifiers.

[66] introduced the *commutativity lattice* for reasoning about commutativity conditions. Specifically, we can build a partially ordered set (poset) $P = (\Psi, \preceq)$ based on logical implication, such that given two commutativity conditions $cSpec_1, cSpec_2 \in \Psi$, $\psi_1 \preceq \psi_2$ iff $\psi_2 \Rightarrow \psi_2$. P has a natural least element, $\bot = \texttt{false}$, and a greatest element, ψ^*, denoting the weakest commutativity condition of a pair of methods. In the literature of optimistic concurrency control, a fine-grained commutativity specification allows more parallelism for transactions to be committed. Analogously, in the settings of stateless model checking, a fine-grained commutativity specification can yield a weaker $\texttt{NC}$ relation between invocations on a semantic execution than a coarse-grained one does. Thus the more precise a commutativity specification is, the coarser the SC-equivalence is induced.

For many frequently used data structures, *e.g.,* set and hashmaps, we do not need to capture their abstract states in the commutativity specifications. For such scenarios, we can drop the S segment in L_1 and replace F with $f(V, V, \ldots)$, *i.e.,* functions can be used over arguments and return values, but not on the abstract states. We refer to this logical fragment as L_2 logic in the remaining of this work. For example, a commutativity specification for *hashmaps* expressed in the L_2 logic is shown in Table 4.2.

Definition 4.3.6 (Commutativity Specifications)**.** Given a set of methods $M \subseteq \mathsf{Method}^L$, a *commutativity specification* S is a tuple $\langle M, \Psi \rangle$, where Ψ is a set of commutativity conditions $\psi^{m_1}_{m_2}(\vec{x_1}; \vec{x_2})$ for all $m_1, m_2 \in M$.

We define the logical *commutativity specification* Ψ for an object with methods $M \subseteq \mathsf{Method}$ as the set of commutativity conditions $\psi^{m_1}_{m_2}(\vec{x_1}; \vec{x_2})$ for all $m_1, m_2 \in M$. We write $S.\Psi$ for the Ψ component of S. We require the commutativity conditions to represent symmetric predicates on two invocations of the same method, *i.e.,* $\psi^{m}_{m}(\vec{x_1}; \vec{x_2}) \equiv \psi^{m}_{m}(\vec{x_2}; \vec{x_1})$ for all $\psi^m_m \in S.\Psi$. For a commutativity condition $\psi^{m_1}_{m_2}(\vec{x_1}; \vec{x_2})$ and invocations u, v with $\mathsf{met}(u) = m_1$ and $\mathsf{met}(v) = m_2$, we write $\psi^{m_1}_{m_2}(v_1, v_2)$ for the substitution of the suitable terms of v_i for $\vec{x_i}$ into the formula $\psi^{m_1}_{m_2}$. This allows us to evaluate the formula in concrete values from v_1, v_2. Given a commutativity specification S, we write $S.\Psi(v_1, v_2)$ for $\psi^{\mathsf{met}(v_1)}_{\mathsf{met}(v_2)}(v_1, v_2)$ where $\psi^{\mathsf{met}(v_1)}_{\mathsf{met}(v_2)} \in S.\Psi$.

Soundness and Completeness. We give a formal definition for the soundness and completeness of commutativity specifications.

Definition 4.3.7 (Soundness)**.** A logical commutativity specification $\langle M, \Psi \rangle$ is sound iff for every $m_1, m_2 \in M$ and two invocations u, v with $\mathsf{met}(u) = m_1$ and $\mathsf{met}(v) = m_2$, we have that $\Psi(u, v)$ implies $u \bowtie v$.

Definition 4.3.8 (Completeness)**.** A logical commutativity specification $\langle M, \Psi \rangle$ is sound iff for every $m_1, m_2 \in M$ and two invocations u, v with $\mathsf{met}(u) = m_1$ and $\mathsf{met}(v) = m_2$, we have that $u \bowtie v$ implies $\Psi(u, v)$.

Remarks. As we will see in §4.4, the semantic completeness of NCMC requires that the commutativity specification S provided by the user is sound. On the other hand, the number of executions explored by NCMC depends on the precision of S. The more precise S is, the fewer number of executions NCMC in principle will explore. Formally, given two commutativity specifications S_1 and S_2 with $S_1 \preceq S_2$, the number of executions explored by NCMC under S_2 is no greater than that under S_1. Our work thus bridges the gap between the work of two communities, namely commutativity analysis in concurrency control and stateless model checking.

4.3.5 The Semantic-Happens-Before $\mathtt{SSH}$ Relation

Definition 4.3.9 (Semantic Extensions)**.** Given a PEG G and a commutativity specification $S = \langle M, \Psi \rangle$, a *semantic extension* of G, denoted as Ext_G, is a tuple $\langle \mathsf{top}, S \rangle$, where $\mathsf{top} : G.\mathtt{E} \rightharpoonup \mathsf{Inv}$ is the partial function that maps each scoped events in $G.\mathtt{E}$ to an invocation of one of the method in M, and maps each unscoped events to $\bot$.

Definition 4.3.10 (Semantic Execution Graph)**.** Given a semantic extension $\mathsf{Ext}_G = \langle \mathsf{top}, S \rangle$, the semantic execution graph (SEG) of Ext_G, denoted as H, is a tuple $\langle E, I, \mathtt{PO}, \mathtt{RF}, \mathtt{IO}, \mathtt{NC} \rangle$, where:

- E is the set of unscoped events of G: $E \triangleq G.\mathtt{E}^u$;
- I is a set of invocations over the methods $S.M$;
- $\mathtt{PO} \subseteq (E \cup I) \times (E \cup I)$ is the *program order* over both unscoped events and invocations;
- $\mathtt{RF} \subseteq (E \cap R) \times (E \cap W)$ is the *reads-from* function over unscoped read and write events in E;
- $\mathtt{IO} \triangleq \{\langle u, v \rangle \mid u, v \in I \wedge (\exists a \in \mathtt{evt}(u), b \in \mathtt{evt}(v) \text{ such that } \langle a, b \rangle \in G.\mathtt{HB})\}$ is the *invocation order*, capturing the dependence between invocations at the level of instructions; and
- $\mathtt{NC} \subseteq \mathtt{IO}$ is the *noncommutative causal* order, defined as the transitive closure of $\{\langle u, v \rangle \mid \langle u, v \rangle \in \mathtt{IO} \wedge \neg S.\Psi(u, v)\}$. Recall from §4.3.4 that the $S.\Psi(u, v)$ function evaluates the commutativity condition on the arguments, return values and abstract states two invocations u and v.

Given a SEG H, we use the '$H.$' prefix to project its components (*e.g.,* $H.\mathtt{PO}$). We define $H.\mathtt{I}_0 \triangleq H.\mathtt{I} \cap \mathsf{Inv}_0$ for the invocations of initialization method (constructors). Similar to PEGs, we define $\mathtt{WO}$ for H as the restriction of $G.\mathtt{WO}$ to unscoped events: $H.\mathtt{WO} \triangleq G.\mathtt{WO}|_{\mathsf{Event}^u}$, and define $\mathtt{RB}$ for H as the restriction of $G.\mathtt{RB}$ to unscoped events: $H.\mathtt{RB} \triangleq G.\mathtt{RB}|_{\mathsf{Event}^u}$.

The $\mathtt{SSH}$ Relation. The $\mathtt{SSH}$ relation denotes the *semantic-happens-before* relation. Intuitively, $\mathtt{SSH}$ captures causality between an unscoped event and an invocation. Given a semantic execution H, we define $H.\mathtt{SSH}$ as follows:

$$H.\mathtt{SSH} = (H.\mathtt{PO} \cup H.\mathtt{RF} \cup H.\mathtt{RB} \cup H.\mathtt{WO} \cup H.\mathtt{NC})^+; [H.\mathtt{I}]$$

Constructing Semantic Executions from Plain Executions. Given a PEG $G = \langle E, \mathtt{PO}, \mathtt{RF}\rangle$ and its semantic extension $\mathsf{Ext} = \langle \mathsf{top}, S\rangle$, we can construct the semantic execution of G, denoted as H, as follows: we first use top to partition $G.\mathtt{E}$ into unscoped events $E^u = \{a \mid a \in E \wedge \mathsf{top}(a) = \bot\}$ and scoped events $E^s = E \setminus E^u$, obtaining $H.\mathtt{E} = E^u$. We then substitute scoped events with its top-level invocation $\mathsf{top}(a)$ for each $a \in E^s$, obtaining $H.\mathtt{I}$, *i.e.,* $H.\mathtt{I} = \{v \mid \exists a \in E$ such that $\mathsf{top}(a) = v\}$. Next, $H.\mathtt{PO}$, $H.\mathtt{RF}$ are obtained as follows:

$$H.\mathtt{PO} = \{\langle e_1, e_2\rangle \mid e_1, e_2 \in H.\mathtt{E} \cup H.\mathtt{I} \wedge \langle \mathsf{rep}(e_1), \mathsf{rep}(e_2)\rangle \in \mathtt{PO}\}$$

$$H.\mathtt{RF} = \mathtt{RF}|_{H.\mathtt{E}}$$

Informally, the SEG H extends PEG G in two aspects. First, given a set of linearizable methods M^L of some data structures, scoped event nodes are merged into a single invocation node. For $\mathtt{PO}$ and $\mathtt{RF}$ edges on scoped events, we also merge them in the following way: if any scoped event $a \in G.\mathtt{E}^u$ is associated with a $\mathtt{PO}$ or $\mathtt{RF}$ edge, then its top-level invocation $\mathsf{top}(a)$ is associated with the corresponding $H.\mathtt{PO}$ or $H.\mathtt{RF}$ edge. Since we assume the client program uses the data structures properly, the writes in these invocations are not observable to the client. This implies that no $\mathtt{RF}$ edge relates a scoped event with an unscoped event. Therefore, we can partition $G.\mathtt{RF}$ into the *unscoped reads-from* relation, $G.\mathtt{RF}^u$, and the *scoped* *reads-from* relation, $\mathtt{RF}^s$; similarly for $G.\mathtt{RB}$ and $G.\mathtt{WO}$. Second, the noncommutative causal order $\mathtt{NC}$ is introduced to characterize the semantically noncommutative orderings between invocations. Intuitively, $\mathtt{NC}$ is a subset of $\mathtt{IO}$ that contributes to the noncommutative causality between invocations. This implies two invocations not related by $\mathtt{IO}$ always commute. In other words, we require that two invocations that are independent at the instruction level are also independent at the semantics level.

We write $\mathsf{Ext}_S^{\mathsf{top}}(G)$ for the semantic execution constructed from G under $\mathsf{Ext} = \langle \mathsf{top}, S\rangle$. Since given a PEG G and a semantic extension $\mathsf{Ext} = \langle \mathsf{top}, S\rangle$, a SEG can be uniquely constructed, in the remaining of this work, we use the tuple $\langle G, \mathsf{top}, S\rangle$ to denote a SEG.

Definition 4.3.11 (Semantic Commutativity Equivalence)**.** Given a PEG G and a semantic extension $\mathsf{Ext} = \langle \mathsf{top}, S\rangle$ and a PEG G, two PEGs G_1 and G_2 are semantically-commutative equivalent (*SC-equivalent*) under $\mathsf{Ext} = \langle \mathsf{top}, S\rangle$, denoted as $G_1 \equiv_{\mathsf{Ext}} G_2$, if $\mathsf{Ext}_S^{\mathsf{top}}(G_1) = \mathsf{Ext}_S^{\mathsf{top}}(G_2)$.

Following the above definition, it is easy to see that SC-equivalence is coarser than the equivalence captured by plain execution graphs, since semantic executions can be viewed as a partitioning of plain executions.

4.4 NCMC: Commutativity-aware Stateless Model Checking

NCMC can be built as an extension of any existing SMC algorithm that considers (only) the independence at the instruction level. We assume the user provides a commutativity specification for data structures in the library code. In this section, we present a version of NCMC that is built on top of *maximal causality reduction* (MCR) [46]. Basing our algorithm on MCR allows us to reason about the soundness and semantic completeness of our algorithm. Furthermore, MCR exploits noncommutative causality between read and write events at the instruction level. Combining the semantic pruning of NCMC and the syntactical pruning of MCR potentially yields more effective reductions in the number of explored executions.

4.4.1 Maximal Causality Reduction

Maximal Causality Reduction (MCR) is a stateless model checking algorithm that covers the full state space of concurrent programs with a *provably minimal* number of executions [46]. Each execution corresponds to a *maximal causal model* [10, 19], which captures the largest possible set of *causally equivalent* executions. MCR can in principle achieve a much coarser partitioning of executions than existing DPOR algorithms. We note that the MCR proposed in [46] is based on interleaving semantics, where executions are represented as traces. Here we present it in the declarative semantics, where executions are modeled as partially ordered graphs.

Intuitively, given one execution of the program, MCR generates a set of constraints, Φ, that allow a particular read in the execution to read a different value. We write $\texttt{feasible}(G)$ to denote the equivalent class of executions that contains the trace G. Given an observed trace G, MCR encodes the constraints of $\texttt{feasible}(G)$ into a logical formula Φ_G. In this paper, we simply write Φ for Φ_G when G is clear from the context.

Each variable in Φ, denoted as O_e, corresponds to the serial order of an event e in the linear extension of an execution $G \in \texttt{feasible}(G)$. Φ is represented as a conjunction of three subformulas, *i.e.,* $\Phi \triangleq \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi rw$, defined in Fig. 4.6.

The Must-happen-before Constraint Φ_{mhb}. The must-happen-before constraint encodes the $\mathsf{MustHappenBefore}$ relation on events. For all pairs of events a, b with $\langle a, b \rangle \in G.\mathsf{MustHappenBefore}|_{\texttt{imm}}$, we use $O_a < O_b$ to encode the program order between a and b.

The Lock-mutual-exclusion Constraint Φ_{lock}. The locking semantics require that two critical sections protected by the same lock do not overlap. We use the lock-mutual-exclusion constraint Φ_{lock} to capture the mutual exclusion semantics over *lock* and *unlock* events. We assume each unlock event on l is paired with the most recent lock event on l in the same thread. We define the 'lock pair on l' relation, $\texttt{LP}_l$ to capture such pairing between the lock

$$\Phi_G \triangleq \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi rw$$

$$\Phi_{mhb} \triangleq \bigwedge_{(a,b)\in G.\mathsf{MustHappenBefore}|_{\mathtt{imm}}} O_a < O_b$$

$$\Phi_{lock} \triangleq \bigwedge_{(a,b),(c,d)\in G.\mathtt{LP}_l} O_b < O_c \vee O_d < O_a$$

$$\Phi_{dc} \triangleq \bigvee_{e\in G.E} \Phi_{rw}(e)$$

$$\Phi_{rw}(e) \triangleq \bigwedge_{\substack{(r,e)\in G.\mathsf{MustHappenBefore}\wedge\\ r\in G.R}} \Phi_{val}(r, val(r))$$

$$\Phi_{val}(r, v) \triangleq \bigvee_{w\in G.W\cap W_{loc(r)}} \Big(\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{\substack{w'\neq w\wedge\\ w'\in G.W\cap W_{loc(r)}}} (O_{w'} < O_w \vee O_r < O_{w'})\Big)$$

Figure 4.6: Constraints encoding for Φ_G.

event and the unlock event on the same critical section. Formally,

$$G.\mathtt{LP}_l \triangleq \{\langle a, b\rangle \mid a \in G.L_l \wedge b \in G.U_l \wedge (\nexists e \in G.L_l \cup G.U_l \text{ such that } \langle a, e\rangle \in G.\mathtt{PO} \wedge \langle e, b\rangle \in G.\mathtt{PO})\}$$

where $G.L_l$ and $G.L_l$ denote the set of lock events and unlock events on lock l, respectively.

The Data-validity Constraint Φ_{dc}. The data-validity constraints ensure that every event in the *considered* execution is feasible. Note that in constructing MCM for an input execution G, the considered execution does not necessarily contain all the events in G but may contain a subset of them, so that all the incomplete executions corresponding to partial executions of the program are considered as well. For an event to be feasible, all the events that must-happen-before it should also be feasible. Moreover, all read events that must-happen-before it should read the *same value* (not necessarily from the *same write*) as that in the input execution; otherwise the event might become infeasible due to a different value read by an event that it depends on.

Algorithm 4 NCMC main procedure

```
1: procedure VERIFY($P, S$)
2:     $H_0 \leftarrow \langle G_0, \mathsf{top}_0, S\rangle$
3:     $\langle H_{\mathtt{pre}}, \Gamma\rangle \leftarrow \langle H_0, \varnothing\rangle$
4:     do
5:         $H \leftarrow$ EXECUTEONE($P, H_{\mathtt{pre}}$)
6:         GENSEEDEXEC($H, \Gamma$)
7:     while $\langle H_{\mathtt{pre}}, \Gamma\rangle \leftarrow \mathtt{nextExp}(\Gamma)$
```

Algorithm 5 Execute P by extending $\mathtt{G_{pre}}$, output the semantic execution

```
1: procedure EXECUTEONE($P, H_{\mathtt{pre}}$)
2:     $\langle G, \mathsf{top}, S\rangle \leftarrow H_{\mathtt{pre}}$
3:     while $a \leftarrow \mathtt{nextEvent_P}(G)$ do
4:         if $a \in \mathtt{error}$ then exit("Errorneous program.")
5:         $\mathtt{Add}(G, a)$
6:         if $a \in \mathtt{R}$ then $G.\mathtt{RF}[a] \leftarrow \mathtt{min}(G.\mathtt{WO}_{\mathsf{loc}(()a)})$        ▷ Update RF
7:         if $v \leftarrow \mathtt{topEvent}(a)$ then
8:             $\mathsf{top}[a] \leftarrow v$
9:             if $a \in \mathtt{invEnd}$ then        ▷ Last event in the current invocation
10:                $U \leftarrow \mathsf{Ext}_S^{\mathsf{top}}(G).\mathtt{I}_{\mathsf{obj}(v)}$        ▷ Get all invocations on the same object
11:                for all $u \in U$ do
12:                    if $\neg S.\Psi(u, v)$ then $\mathsf{Ext}_S^{\mathsf{top}}(G).\mathtt{NC}[u] \leftarrow v$        ▷ Update NC
13:    return $\mathsf{Ext}_S^{\mathsf{top}}(G)$
```

4.4.2 The NCMC Algorithm

The Main VERIFY Procedure Algorithm 4 outlines the main algorithm. Given a commutativity specification S, NCMC begins exploring the executions of a program P by calling VERIFY(P, S). This procedure creates an initial semantic execution prefix $H_0 = \langle G_0, \mathsf{top}_0, S\rangle$ (H_0 consists of an empty seed execution containing only initialization events and an empty top function) and an empty work set Γ. It then generates executions of P one at a time by calling EXECUTEONE at line 4, which fully explores one execution extending $H_{\mathtt{pre}}$, and builds a PEG G and its corresponding SEG H. Then the GENSEEDEXEC(H, Γ) procedure generates a set of seed executions from H, and pushes them to the work set Γ.

Algorithm 6 Generate seed executions from H and add them to Γ.

```
 1: procedure GenSeedExec($H, \Gamma$)
 2:     $\langle G, \mathsf{top}, S\rangle \leftarrow H$
 3:     for all $r \in H.\mathtt{R}$ do
 4:         $\mathtt{expVals}_r \leftarrow \{\mathsf{val}(r)\}$
 5:         $W \leftarrow G.\mathtt{W}_{\mathsf{loc}(()r)}$
 6:         if $\mathsf{top}(r) = \bot$ then                                  ▷ Handle unscoped reads
 7:             for all $w \in W \land \mathsf{val}(w) \notin \mathtt{expVals}_r$ do
 8:                 $G_{\mathtt{pre}} \leftarrow$CheckRW$(r, w, \mathtt{expVals}_r)$
 9:         else                                                     ▷ Handle scoped reads
10:             for all $w \in W \land \mathsf{val}(w) \notin \mathtt{expVals}_r$ do
11:                 $E \leftarrow H.\mathtt{SSH}^{-1}[\mathsf{top}(r)] \cap H.\mathtt{PO}[\mathsf{top}(w)]$
12:                 $G_{\mathtt{pre}} \leftarrow$CheckRWComm$(r, w, E, \mathtt{expVals}_r)$
13:         if $G_{\mathtt{pre}} \neq \bot$ then push$(\Gamma, \langle G_{\mathtt{pre}}, H.\mathsf{top}, S\rangle)$     ▷ Update the work set
```

The `nextExp` procedure pops $H_{\mathtt{pre}}$, the next seed execution to be explored, from the work set Γ.

The ExecuteOne Procedure The ExecuteOne procedure in Algorithm 5 explores one execution at a time and constructs the corresponding semantic execution on the fly. Given a seed semantic execution $H_{\mathtt{pre}}$, NCMC first obtains its components $G_{\mathtt{pre}}$ and top, where $G_{\mathtt{pre}}$ is the prefix of the plain execution. Next, NCMC extends the plain execution graph by its next event a, given by the $\mathtt{nextEvent}_P(G)$ function. If $\mathtt{nextEvent}_P(G)$ returns $\bot$, the program has either terminated or all threads are blocked. In this case, the ExecuteOne procedure returns and outputs the semantic execution graph. If the next event is an assertion violation, then an error is reported and the algorithm terminates. Otherwise, the algorithm extends G by the next event a, and update $G.\mathtt{RF}$ if a is a read event. Next, if a belongs to some invocation v, the algorithm adds the mapping from a to v to top. If a is also the end of the invocation v, the algorithm enumerates all previous invocations on the same object and evaluates the commutativity condition between v and each previous invocation. If the commutativity condition evaluates to `false`, the algorithm adds an $\mathtt{NC}$ edge between the noncommutative pair of invocations.

The functions `topEvent` can be implemented by instrumenting the source program. When a method of interest is invoked, the algorithm attaches additional labels to all events within the invocation. This can also tell us when a primitive event is the last event of an invocation.

This procedure improves the original online execution procedure of MCR in two aspects. First, it builds the PEG online in addition to collect a trace, which enables an efficient algorithm to construct the MCM constraints. Second, a SEG of the current execution is constructed on the fly. As is discussed earlier, the semantic execution graph is the crux for exploiting semantic commutativity in the model checking algorithm. The online semantic execution construction algorithm can be potentially applied to other concurrency analyzers, *e.g.,* commutativity race detection. Hence, the algorithm presented in this section is of interest beyond stateless model checking.

The GenSeedExec Procedure The GENSEEDEXEC procedure in Algorithm 6 is the key to yielding effective reductions in the number of explored executions. Essentially, the algorithm works by pivoting around the value of reads in the execution; to generate a new seed execution, we make a read event in the current execution to read a different value from another write, if feasible. In particular, we encode such feasibility conditions into logical formulas and query an SMT solver for satisfiability.

This new GENSEEDEXEC procedure is different from the previous algorithm in [46] in that it can exploit semantic commutativity to avoid constructing redundant seed executions. Specifically, NCMC handles scoped reads and unscoped reads differently. The handling of scoped reads carries the crux of pruning SC-equivalence executions, described at line 7-10.

Handle Unscoped Reads. We first describe the handling of unscoped reads (line 1-6). Given an execution G, our algorithm enumerates each unscoped read r in G, and uses the set $\texttt{expVals}_r$ to record values that have been explored so far, which is initialized to contain $\textsf{val}(r)$ only. For each write w that writes to the same location as r and whose value is not in $\texttt{expVals}_r$ (*i.e.,* not explored yet), our algorithm proceeds by calling the CHECKRW procedure, which constructs a logical formula $\Phi_{r,w}$ that constrains r to read from w, defined as follows:

$$\Phi_{r,w} \triangleq \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{rw}(r) \wedge \Phi_{rw}(w) \wedge \Phi_{val}(r, val(w))$$

where $\Phi_{val}(r, v)$ denotes the constraint for r to read a value v, as is defined in Section 4.4.1. The algorithm then invokes an SMT solver for $\Phi_{r,w}$. If $\Phi_{r,w}$ is satisfiable, we extract a new execution prefix from the solution and pushes it to the work set Γ along with its semantic extension $\langle H.\mathsf{top}, H.\mathtt{NC}\rangle$. In addition, the value of w is added to the set $\mathtt{expVals_r}$ to avoid redundant explorations in the future.

Handle Scoped Reads. We now discuss the handling of scoped reads (line 7-10). Again, our algorithm enumerates each scoped read r in execution G, and deals with a write w that writes to the same location as r if r has not read the value of w yet. Our algorithm proceeds by checking if there is a node that is both PO-ordered after w and SSH-ordered before r. The set of such events E is captured by $H.\mathtt{SSH}^{-1}[\mathsf{top}(r)] \cap H.\mathtt{PO}[\mathsf{top}(w)]$. If there is none, making r read from w, if feasible, results in a seed execution that is SC-equivalent to the current one's prefix. In this case, the algorithm skips making r to read from w. Otherwise, to generate a seed execution that is not SC-equivalent to the current one, we must ensure that the invocation that contains r, $\mathsf{top}(r)$, is invoked before at least one of these primitive events. Specifically, we capture this constraint with the logical formula $\Phi^{nc}_{r,w,E}$ defined as follows:

$$\Phi^{nc}_{r,w,E} \triangleq \bigvee_{e \in E} (O_e < O_r \wedge \Phi_{rw}(e))$$

The formula $\Phi^{nc}_{r,w,E}$ ensures that at least one event in E needs to happen before r in the generated seed execution, if feasible. The CheckRWComm procedure is different from CheckRW in that it builds a formula that is a conjunction of Φ_{rw} and $\Phi^{nc}_{r,w,E}$, ensuring the newly generated seed execution to be semantically different from the correct one.

Example. The idea of using the $\Phi^{nc}_{r,w,E}$ constraint to prune redundant executions can be illustrated in the following example. Consider the following program where c is a counter

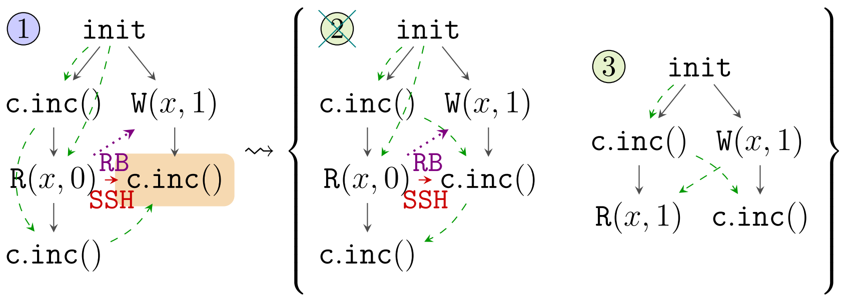


Figure 4.7: Key exploration steps COUNTER-SHB2, with NC edges omitted for brevity.

object implemented similarly as the previous examples:

$$\begin{array}{l||l} c.inc(); & \\ a := x; & x = 1; \\ & c.inc(); \\ \textbf{if } a = 0 \textbf{ then } c.inc(); & \end{array} \qquad \text{(COUNTER-SHB2)}$$

Consider the case when NCMC arrives at execution ① in Fig. 4.7. For brevity, NC edges are omitted. The SSH edge is due to the RF between R$(x, 0)$ and the PO between W$(x, 1)$ and $c.inc()$. Now NCMC begins exploring reads-from options for the read r in the highlighted $c.inc()$ node. In particular, suppose the next reads-from option is the write w in the first $c.inc()$ in the first thread. Without the $\Phi^{nc}_{r,w,E}$ constraint, NCMC may explore both executions ② and ③, where the execution ② is SC-equivalent to ① and is thus redundant. The $\Phi^{nc}_{r,w,E}$ constraint ensures that r is ordered before the node R$(x, 0)$ in the new seed exploration, if feasible, resulting in only execution ③ being explored. NCMC guarantees that any generated seed exploration is not SC-equivalent to the current execution.

4.4.3 Soundness and Semantic Completeness

In the rest of this section, we establish the correctness of NCMC by showing that the executions explored satisfy two properties: *soundness* and *semantic completeness*. Given a program P, we say that NCMC generates execution G for P if running VERIFY(P) outputs G among other executions. Soundness ensures that if NCMC generates G, then G is feasible; semantic completeness ensures that if G is a consistent execution of P, then NCMC generates at least one execution G' that is SC-equivalence to G for P with respect to the given commutativity specification.

Soundness The soundness of NCMC follows immediately from that of MCR. Specifically, for a given execution G and a particular read r in G, NCMC handles the exploration of a new value for r in the following three cases:

1. r is an unscoped read, which is explored in the same way as that of MCR;

2. r is a scoped read and $H.\texttt{SSH}^{-1}[\mathsf{top}(r)] \cap H.\texttt{PO}[\mathsf{top}(w)]$ is empty, meaning the new read-from relation to be enforced, if feasible, does not lead to a seed execution that is SC-equivalent to the current one's prefix.

3. r is a scoped read and $E = H.\texttt{SSH}^{-1}[\mathsf{top}(r)] \cap H.\texttt{PO}[\mathsf{top}(w)]$ is not empty. In this case, NCMC adds additional constraints $\Phi^{nc}_{r,w,E}$, which forces the generated seed execution to be semantically different than the current one.

In all three cases, since the constraint encoding is sound (following the soundness of MCM encoding), the seed executions is feasible. Since NCMC extends a seed execution $G_{\texttt{pre}}$ by continuing exploring events following $G_{\texttt{pre}}$, the resulting G extended from $G_{\texttt{pre}}$ is feasible.

Semantic Completeness. Establishing the semantic completeness of NCMC, however, is more involved. Let us first introduce some definitions that will be used for the proof.

Definition 4.4.1 (Configurations of NCMC)**.** A configuration C of NCMC is a tuple $\langle H, \Gamma \rangle$, where H is the current semantic execution and Γ is the work set from which $H_{\texttt{pre}}$ is just popped, *i.e.,* before adding any seed executions to Γ.

Definition 4.4.2 (Post-images of a Seed Execution)**.** The post-image of a seed execution $G_{\texttt{pre}}$, denoted as $\texttt{post}(G_{\texttt{pre}})$, is the set of possible executions reachable from $G_{\texttt{pre}}$.

We first prove that every execution pruned by NCMC is SC-equivalent to the current execution explored by NCMC.

Theorem 1 (No Redundancy of NCMC). Given a configuration $C = \langle H, \Gamma \rangle$ with $H = \langle \mathsf{top}, S \rangle$, for any execution prefix $H_{\mathtt{pre}} = \langle G_{\mathtt{pre}}, \mathsf{top}, S \rangle$ generated at C, $G_{\mathtt{pre}} \not\equiv_{\mathsf{Ext}} G$.

Proof sketch. Consider two cases for the newly enforced reads-from pair $\langle r, w \rangle$ that underpins $G_{\mathtt{pre}}$.

Case 1. r is unscoped. By the construction of $\Phi_{r,w}$, for all $G' \in \mathtt{post}(G_{\mathtt{pre}})$, $\langle r, w \rangle \in G'.\mathtt{RF}$. On the other hand, $\langle r, w \rangle \notin G.\mathtt{RF}$. We have $G' \not\equiv_{\mathsf{Ext}} G$ since $H.\mathtt{RF} \neq H'.\mathtt{RF}$ since r, w are unscoped.

Case 2. r is scoped. Following algorithm 6 and the construction of $\Phi_{r,w} \wedge \Phi^{nc}_{r,w,E}$, there exists $e \in H.\mathtt{SSH}^{-1}[\mathsf{top}(r)] \cap H.\mathtt{PO}[\mathsf{top}(w)]$ such that for all $G' \in \mathtt{post}(G_{\mathtt{pre}})$, $e \in G'$ and $O_e < O_r < O_w$. If e is unscoped, $\mathsf{Ext}^{\mathsf{top}}_S(G).\mathtt{SSH} \neq \mathsf{Ext}^{\mathsf{top}}_S(G').\mathtt{SSH}$ and we are done. If e is scoped, assume in contradiction that $G \equiv_{\mathsf{Ext}} G'$. Let $u = \mathsf{top}(e), v = \mathsf{top}(r)$. Then u, v are also in $\mathsf{Ext}^{\mathsf{top}}_S(G).\mathtt{I}$, and $\langle u, v \rangle \notin \mathsf{Ext}^{\mathsf{top}}_S(G).\mathtt{SSH}$. However, this contradicts with the satisfiability of $\Phi^{nc}_{r,w,E}$, which implies that $\langle u, v \rangle \in \mathsf{Ext}^{\mathsf{top}}_S(G).\mathtt{SSH}$. $\square$

Semantic completeness follows immediately from no-redundancy of NCMC and completeness of MCR. The executions pruned by NCMC at a configuration $\langle G, \Gamma \rangle$ is SC-equivalent G. This implies that given an execution G' generated by MCR, NCMC does not generate it iff G' is SC-equivalent to G for some configuration $\langle G, \Gamma \rangle$. We summarize the correctness of NCMC in the following theorem.

Theorem 2 (Correctness). The NCMC algorithm is sound. If the commutativity specification used is sound, then NCMC is semantically complete, and at any configuration, NCMC generates no executions that are SC-equivalent.

4.5 Evaluation

We focus on answering the following two questions in our evaluation of NCMC:

1. How is the *performance* of our approach compared with other state-of-the-art model checkers?

2. How does the precision of commutativity specifications affect the performance of NCMC?

To evaluate the performance of our approach, we have implemented NCMC as a verification tool for multithreaded Java programs, based on the open-source JMCR model checker [46], available at `https://github.com/parasol-aser/JMCR`. In addition to comparing with JMCR as a baseline, we also compare NCMC against Yogar-CBMC, a static model checker for C programs that has won the first place in the concurrency safety category of TACAS SV-COMP for the year 2017-2019, available at `https://github.com/yinliangze/yogar-cbmc`.

4.5.1 Evaluation Methodology

Standard Benchmarks. We first evaluate NCMC on a set of benchmarks taken from the TACAS competition on Software Verification [2]. These benchmarks have been used extensively in the SMC literature by many tools [43, 67, 68].

We use a set of benchmarks under the concurrency safety category that have critical sections convertible to method invocations on data structures. The purpose of this benchmark set is to evaluate the overall performance of our algorithm on a standard test suite, and also to show that the overhead induced by constructing SEGs and evaluating commutativity conditions between method invocations (where NCMC may not bring much improvement over the underlying MCR algorithm).

Benchmark Conversion. We convert these C programs into Java, since our tool only supports Java programs. To keep the semantics of the converted Java programs as close as possible to the corresponding C programs, we simulate RMW instructions with multiple read and write instructions protected by locks. For some of these benchmarks, we refactored the code into object-oriented style so that we can define commutativity conditions on the methods of these objects. For instance, in the benchmark `pthread-demo-datarace-1`, we encapsulate the code $lock(l); x = x + 1; unlock(l)$ into a method, $increase()$, defined on a $Counter$ object.

We note that our conversion can potentially lead to different behaviors of the benchmarks used by NCMC and Yogar-CBMC at the instruction level (*i.e.,* their state spaces are incomparable). However, apart from differences in the language features and nuances, our conversion is faithful in keeping the semantics of the programs and preserves the inherent state-space complexity of each benchmark. As such, we believe that our comparison of NCMC and Yogar-CBMC on a set of different but functional equivalent benchmarks still gives valuable insight in the relative performance of NCMC against other state-of-the-art model checkers.

Data Structure Benchmarks. We then evaluate NCMC on a collection of data structure benchmarks, each given a coarse-grained and a fine-grained commutativity specifications, under two different workloads. The data structures used include sets (implemented as a linked list), hashtables and union-find. A union-find data structure represents a set of disjoint sets, and supports `union` and `find` methods. The `find` method is implemented with path compression[2]. These data structures have frequently appeared in various contests of commutativity, including speculative execution, verification and generation of commutativity conditions [66, 65, 50, 51]. Our first workload includes only querying, and the second includes both querying and updating. All data structures use coarse-grained locking for each method, allowing us to treat them as atomic invocations.

The purpose of this experiment is to provide a more realistic workload for evaluating the performance of NCMC, under different precisions of commutativity specifications. Our results show that NCMC achieves exponential reductions not only for the first set of standard benchmarks, but also for these more realistic workloads, and is able to explore exhaustively behaviors of client programs of such data structures significantly faster than the techniques that are oblivious to semantic commutativity. In addition, as we will see in §4.5.3, precision of the commutativity specifications affects the performance of NCMC, depending on the

[2]Path compression flattens the structure of the tree by making every node point to the root whenever `find` is used on it. The concrete state of the data structure can change due to the side effect of pass compression, while the abstract state of it does not change.

actual workload.

Commutativity Specifications Used. For the benchmarks from [2], we manually inspected the code of each benchmark and wrote the commutativity specifications for them. We note that some of these benchmarks do not define atomic methods but have critical sections instead. To exploit the commutativity conditions on them, we convert critical sections that are conditionally commutative to methods where applicable. All commutativity specifications except the one for `workstealqueue_mutex` are in L_2 logic that only involves the method call arguments for each pair of methods. For `workstealqueue_mutex`, the commutativity conditions involve the abstract states of objects.

For the data structure benchmarks, the commutativity specifications are in L_1 logic for sets and hashtables, and are in L_2 logic for union-find data structures. For each data structures, we provide two versions of commutativity specifications, a fine-grained one and a coarse-grained one. We compare the relative performance on different selections of commutativity specifications in §4.5.3.

Experimental Setup. We conducted all experiments on a 64-bit MacOS with an Intel Core i7 CPU (dual cores@2.7GHz) and 16GB of RAM. All reported times are in seconds unless explicitly noted otherwise. We set the timeout limit for each benchmark run to 15 minutes.

4.5.2 Standard Benchmarks

In our first experiment, we compare NCMC with Yogar-CBMC and JMCR on standard benchmarks from [2], with their results summarized in Table 4.3, where LB denotes the loop bound. The metrics used in Table 4.3 are the number of executions explored for JMCR and NCMC, and the time taken for all the three tools (since Yogar-CBMC is a static symbolic execution-based verifier and does not explore dynamic executions, we only show its time taken). The $\otimes$ marks a wrong output from the model checker.

As shown in Table 4.3, in most of these data-intensive benchmarks where complex data

		Executions		Time (s)		
	LB	JMCR	NCMC	Yogar	JMCR	NCMC
sigma	16	1	1	17.34	**0.12**	0.13
workstealqueue_mutex-2	4	768	**693**	**18.89**	22.20	21.08
fkp2013-1	50	-	**52**	149.54	-	**4.69**
pthread-demo-datarace-1	20	-	**1**	305.42	-	**0.361**
qw2004_variant	10	-	**23**	71.26	-	**1.28**
ring_2w1r-1	8	-	**812**	$\otimes$	-	**43.76**

Table 4.3: Benchmarks from SV-COMP [2]. The `LB` column specifies the loop bound used.

structures are involved, the performance of NCMC is better than that of both Yogar-CBMC and JMCR. In most cases, NCMC takes significantly less time to finish. In other cases, NCMC achieves comparable performance with the other tools.

Let us first focus on the two benchmarks where JMCR does not timeout. In the benchmark `sigma`, both JMCR and NCMC take significantly less time than Yogar-CBMC. This is because both JMCR and NCMC are able to hit one assertion error in the benchmark in the first execution, whereas Yogar-CBMC, a SAT-based static model checker with bit-vector-level precision, fails to solve the monolithic encoding of the verification constraints of the program in a short amount of time. Both JMCR and NCMC explore only one execution, while JMCR takes slightly less time. The difference between JMCR and NCMC in this case is due to the additional overhead induced by tracking the `NC` order. In the benchmark `workstealqueue_mutex`, NCMC is slower than Yogar-CBMC but is in the comparable range. On the other hand, by exploiting commutativity of invocations in this benchmark to a certain degree, NCMC explores fewer executions than JMCR and is thus faster than JMCR.

For the rest of the benchmarks, NCMC is able to outperform the other tools by a very large margin. The reason behind this is that all these benchmarks exhibit intensive conflicting memory accesses from multiple threads, but at the semantics level the critical sections that enclose these accesses commute at a very high frequency. In addition, the commutativity conditions on these critical sections can be precisely captured by L_2 logic, allowing NCMC to prune a large amount of seed interleavings to explore with minimal overhead. Finally, in the `ring_2w1r` benchmark, Yogar-CBMC outputs a false violation of the assertions, while NCMC is able to exhaustively explore all the abstract states of the program and terminate successfully.

In summary, NCMC is able to exploit the commutativity of complex critical sections on data-intensive programs with simple commutativity conditions, and can therefore be significantly faster than tools that reason about conflicts at the concrete memory level. In addition, the overhead for evaluating commutativity conditions is relatively small. We note that in most of these benchmarks the commutativity conditions on the critical sections can be captured in relatively simple logic. One exception is `workstealqueue_mutex`, in which we used the abstract states. In the same benchmark, we also observed that the commutativity conditions are not satisfied for most of the method invocations, which may explain why NCMC only achieves a relatively slight reduction, compared with the results in other benchmarks. Nevertheless, NCMC still gains benefit from exploiting commutativity of invocations in this case. As we will see in Section 4.5.3, in data structures with more complex commutativity specifications, the benefits of leveraging commutativity conditions become even more prevalent.

4.5.3 Data Structure Benchmarks

In our second experiment, we evaluate the performance of NCMC against JMCR on data structure benchmarks, and also evaluate how the precision of commutativity specifications affects the performance of NCMC. The results for querying-only workload are shown in Table 4.4; the results for mixed (involving both querying and updating) workload are shown in Table 4.5. Entries with a dash "-" mark that the tool timeout after 15 minutes.

Workload with Query Only. For the querying workload, JMCR and NCMC explore exactly one execution on the set and hashtable benchmarks, regardless of the precision of

	Executions			Time (s)		
	JMCR	$\texttt{NCMC}_\texttt{c}$	$\texttt{NCMC}_\texttt{f}$	JMCR	$\texttt{NCMC}_\texttt{c}$	$\texttt{NCMC}_\texttt{f}$
set(4)	1	1	1	**0.04**	0.27	0.29
set(5)	1	1	1	**0.04**	0.29	0.29
set(6)	1	1	1	**0.05**	0.29	0.30
set(7)	1	1	1	**0.09**	0.38	0.39
hashtable(4)	1	1	1	**0.01**	0.08	0.12
hashtable(5)	1	1	1	**0.02**	0.09	0.12
hashtable(6)	1	1	1	**0.02**	0.10	0.14
hashtable(7)	1	1	1	**0.06**	0.16	0.19
union-find(4)	32	**4**	**4**	1.67	**0.41**	0.65
union-find(5)	64	**5**	**5**	9.60	**0.62**	0.98
union-find(6)	256	**6**	**6**	36.03	**1.08**	1.52
union-find(7)	512	**7**	**7**	74.43	**1.95**	2.61

Table 4.4: Data structure benchmarks with coarse-grained ($\texttt{NCMC}_\texttt{c}$) and fine-grained ($\texttt{NCMC}_\texttt{f}$) commutativity specifications (seekers workload).

the commutativity specifications. The reason is that with no updating in these benchmarks, all queries are deemed commutative at both the instruction level and the semantics level. Although these invocations may be potentially protected with the same locks, their order is not tracked. Thus one execution is sufficient for these benchmarks. One observation here is that JMCR is slight faster than NCMC due to the absence of constructing SEGs and evaluating commutativity consitions.

For the union-find benchmark, NCMC is able to outperform JMCR by a large magnitude. Specifically, NCMC yields exponential reductions on the explored executions. As discussed earlier, two invocations of `find` with path compression may be conflicting at the level of concrete states, but they always commute at the level of semantics. NCMC is able to exploit such semantics commutativity with both commutativity specifications. Another important observation here is that due to the complexity of a precise union-find commutativity specification, $\texttt{NCMC}_\texttt{f}$ is slower than $\texttt{NCMC}_\texttt{c}$ even though they explore the same number of executions. It may seem unnecessary to use a complete commutativity specification for union-find in

	Executions			Time (s)		
	JMCR	$\texttt{NCMC}_\texttt{c}$	$\texttt{NCMC}_\texttt{f}$	JMCR	$\texttt{NCMC}_\texttt{c}$	$\texttt{NCMC}_\texttt{f}$
set(4)	306	**5**	**5**	19.477	**0.26**	**0.26**
set(5)	2809	**6**	**6**	181.226	**0.27**	0.31
set(6)	-	**7**	**7**	-	**0.31**	0.38
set(7)	-	**8**	**8**	-	**0.34**	0.40
hashtable(4)	76	26	**25**	4.71	2.80	**2.58**
hashtable(5)	455	109	**85**	30.97	12.22	**8.87**
hashtable(6)	-	2476	**940**	-	422.34	**224.75**
hashtable(7)	-	-	-	-	-	-
union-find(4)	44	30	**26**	1.67	0.82	**0.70**
union-find(5)	119	46	**42**	4.46	**1.86**	3.56
union-find(6)	740	234	**218**	25.93	**8.48**	10.16
union-find(7)	6400	2466	**2228**	190.08	**86.80**	104.43

Table 4.5: Data structure benchmarks with coarse-grained ($\texttt{NCMC}_\texttt{c}$) and fine-grained ($\texttt{NCMC}_\texttt{f}$) commutativity specifications (mixed workload).

this case, however, as we will see in Table 4.5, the cost introduced by a more complex but weaker commutativity specification pays off when the workload is mixed with both queried and updates.

Workload with Both Query and Update. For the mixed workload (Table 4.5) involving both querying and updating the abstract states of data structures, NCMC with both the coarse-grained ($\texttt{NCMC}_\texttt{c}$) and fine-grained ($\texttt{NCMC}_\texttt{f}$) commutativity specifications maintains a very good performance. For most of these benchmarks, NCMC is able to explore far fewer executions and is thus significantly faster than the commutativity-agnostic JMCR. Similar to the query-only workload, the precision of commutativity specifications can have a great impact on the performance of NCMC. For the set benchmarks, both versions of commutativity specifications yield exponential reduction in the number of executions compared to JMCR and explore exactly the same number of executions. This is due to the structure of the programs. Since each thread contends to add different elements to the set, these insertions commute at both the coarse-grained and fine-grained specification. However, $\texttt{NCMC}_\texttt{f}$ is slower than $\texttt{NCMC}_\texttt{c}$ due to its tracking of more information to utilize a fine-grained commutativity specification. For the hashtable benchmarks, threads can concurrently add elements with the same key to the hashmap, resulting in noncommutative orders enforced on these invocations. Since the fine-grained commutativity specification is weaker and thus allows more parallelism on these insertions, it leads to better performance than the coarse-grained one. Similarly, for union-find, the fine-grained commutativity specification yields better reduction. One interesting observation here is that the time taken per execution for $\texttt{NCMC}_\texttt{f}$ is longer than $\texttt{NCMC}_\texttt{c}$, owing to additional overhead induced by tracking the states of the union-find set. The gains earned by weaker commutativity conditions outweighs the overhead for utilizing these conditions in this case.

Finally, the number of explored executions grows relatively fast even for $\texttt{NCMC}_\texttt{f}$ on the hashtable and union-find benchmarks. This is due to the fact that the number of collisions of noncommutative invocations grows non-linearly with respect to the number of threads. For the hashtable benchmark with 7 threads, $\texttt{NCMC}_\texttt{f}$ does not terminate within 15 minutes, due to a dramatic increase in collisions in the hashmap entries, and thus introducing too many read and write events even for $\texttt{NCMC}_\texttt{f}$ to handle. Nevertheless, NCMC still gains significant improvement in reduction by exploiting commutativity.

4.6 Conclusion

We have presented NCMC, a novel stateless model checking technique that is capable of exploiting semantic commutativity for verifying concurrent programs. By lifting the execution graph defined at concrete events to semantic invocation events, NCMC constructs the semantic execution graph to capture the dependence between method invocations, and avoids exploring redundant executions characterized by reads-from relations that lead to the same abstract states. We have implemented NCMC for verifying multithreaded Java programs and evaluated its performance on a variaty of standard and data structure benchmarks. Our evaluation results show that NCMC yields exponential reductions on the number of executions explored, achieving significantly better or comparable performance than the state-of-the-art SMC algorithms that are oblivous to program semantics including JMCR and Yogar-CBMC.

5. RELATED WORK

There is a large body of work on detecting concurrency related errors. To our knowledge, OVPredict is the first partial-order-based analysis targeting both races and non-racy order violations.

Order Violation Detection. There is a large body of work in finding atomicity violation, order violation, and thread safety violation bugs [69, 70, 71, 34, 72, 73, 74, 15]. These techniques can detect some OV bugs that OVPredict targets, and work by actively inserting delay into target programs to expose atomicity violation interleavings. These techniques typically require multiple runs to profile events, or persist state between runs. In contrast, OVPredict aims to predict more concurrency bugs in one single execution by extracting partial orders from execution and does not persists state between runs.

Predictive Analyses. Predictive analyses aim to infer program behaviors in feasible reorderings of a given trace, and is primarily applied to race detection. Predictive analysis for concurrent UAF detection is first employed in UFO [7], which can potentially detect all concurrent UAF errors knowable from a trace, including non-racy ones. UFO first records an execution trace of a program and then encodes the feasibility constraints of a concurrent UAF into SMT formulas for *postmortem* analysis. As compared in §4.5, UFO cannot analyze long program traces in reasonable time, and resorts to breaking the traces into bounded windows of executions, missing predictive UAF races where the use-free pair is long-distance apart. In contrast, OVPredict can handle executions with billions of events *on the fly*, and works well even for long-running programs.

Our approach is largely inspired by the partial order based race detection techniques. The CP [17] and WCP [20] analyses are sound but miss predictable races. The WDC analysis can report more data races than CP and WCP analyses but is unsound in general. The vindication algorithm of WDC analysis can soundly determine if a WDC race is real but is incomplete [21].

Recently, partial orders incorporating control-flow information have been proposed and can detect races beyond those knowable from the observed trace [22]. M2 [24] proposes a sound predictive race detection algorithm including a phase similar to the WDC vindication. M2 achieves a higher degree of completeness without sacrificing soundness, compared with WCP and WDC analyses, but requires multiple passes of the input trace, which is unsuitable for online race detection.

SMT-based predictive analyses [10, 11, 18] can also incorporate control-flow information. However, they are unscalable and miss races when coupled with a windowing strategy.

Stateless Model Checking. Stateless model checking has been an active research area since the pioneering work of VeriSoft [41]. Since then a large effort has been invested in reduction techniques to combat the explosion of interleaving space. DPOR algorithms that use equivalence partitioning have been proposed for sequential consistency [43, 67], weak momory models [75, 44], and for release-acquire semantics [68]. Different from DPOR, Maximal Causality Reductions (MCR) [46] uses a maximal causal model to partition the state space and has been applied to weak memory models as well [47]. Recently, execution graph-based DROP algorithms [48, 63] have been successfully applied to memory model-agnostic settings. However, all these approaches rely on independence at the instuction level, and are not able to exploit commutativity at the semantics level.

Semantic pruning in DPOR have also been investigated in [45], which uses net unfoldings to achieve reductions on the executions. The most related work to ours is CDPOR [49], a DROP algorithm that exploits *conditional independence* between critical sections. CDPOR uses static analysis to generate independence conditions (ICs), constraints that encode the independence conditions between critical sections, and evaluates these independence conditions during online exploration. However, CDPOR can only exploit critical sections on the concrete states. Our utilization of commutativity specifications can capture commutativity on the abstract states. In addition, our approach exploits commutativity between method invocations, and thus is able to use logic fragments with more expressive power. We note that the two techniques are complementary and can be combined to potentially achieve even better reductions.

Commutativity Analysis. Commutativity analysis has been successfully applied to areas such as optimistic concurrency control [66, 76, 77, 78], code parallelization [79] and proving concurrent programs [80]. The work of [81] uses commutativity specifications to reduce the number of executions explopred for testing the atomicity of client code composing scalable concurrent operations. Their technique is based on modular testing of client code in the precence of an adversarial environment. In our work, we assume the operations on the data structures whose commutativity specifications are provided are atomic, and use them to reduce the number of explored executions for the data structure clients. Therefore our work and theirs are complementary and can be combined; it is possible to first use their method for testing the atomicity of the data structure, and then provide the commutativity specifications on the data structures to NCMC for model checking all possible behaviors of the client code.

Our approach of decoupling the model checking of data structure clients from the correctness of data structure implementations aligns with the theme in [65]. Their work focuses on the verification of sound and complete commutativity conditions of linked data structures. Our work also serves as a motivation for providing correct commutativity specifications.

Commutativity has also been applied to find *commutativity races* [39], an interfering phenomenon caused by incorrect usage of library interfaces that can be viewed as a more general form of data race detection. Similar to their approach, NCMC also analyzes conflicting relations (`NC`) between invocations during online execution of the program. However, our algorithm focuses on model checking, which is also capable of detecting commutativity races. Our work can be viewed as a generalization of their work to the settings of stateless model checking.

Finally, various recent works focus on automatic learning of commutativity specifications [51, 50]. Our work is the combination of commutativity analysis and stateless model checking. In particular, we introduce the notion of semantic execution graphs, while prior work in optimistic transaction control only models executions as a sequence of atomic operations. Our work is the first step towards bridging the gap between the two communities and we plan to investigate this direction further in future work.

We also note that our algorithm can be extended to model checking *transactional programs* under the sequential consistency memory model. The principle of using commutativity conditions to detect conflicts between transactions can be naturally applied to model checking transactional programs. Since serializability requires that concurrent transactions appear to execute one after the other in a total sequential order, we can model each transaction as code blocks protected by locks. More specifically, we can replace each transaction of the form $[T]$ in the transactional program with $lock(l); T; unlock(l)$, where T denotes the instructions to be executed atomically in the transaction. As such, NCMC can expoit commutativity conditions between transactions to reduce the number of executions to be explored.

UAF Detection A large number of UAF detection tools have been developed in recent years. Static analysis has been used to detect sequential as well as concurrent UAF errors [36]. Due to undecidability, they are either unsound or incomplete. *Evidence-based* dynamic UAF detection techniques have been gaining popularity in recent years [26, 82, 83, 84]. Compared to these techniques, OVPredict is distinguished by its ability to predict unseen high-level races from the observed execution.

Schedule Space Exploration. Orthogonal to our work, systematic testing enables more program states to be covered using either *systematically exploration* (*i.e.,* model checking) [85, 46, 18, 86] or heuristics [87, 88, 89, 14]. Combining systematic testing with predictive analyses such as ours is a promising direction to find more bugs.

Static Analysis. One limitation of OVPredict is the reliance on user provided annotations to detect high-level races in libraries. One possible approach to alleviate the annotation efforts is to automatically discover the invocation order constraints on library APIs using static analysis [90, 91].

6. CONCLUSION

This thesis formalizes the problem of commutativity order violation detection by extending the traditional notion of predictable races. The notion of commutativity order violation covers a larger class of concurrency errors that arise in practice. Our technique, OVPredict, detects both racy and non-racy order violations. The latter case is beyond existing race detection based techniques. In our evaluation, OVPredict uncovers several previously unknown order violation errors in open-source Go programs, including one non-racy order violation in CockroachDB. On C and Chromium benchmarks, OVPredict exposes the known UAF errors with higher frequency compared to ThreadSanitizer. Furthermore, it has comparative performance overhead compared the highly optimized HB analysis.

Our work advances the state of the art in predictive analysis by incorporating dependency at the library interface level, which is beyond the basic read-write conflicts that existing race detectors consider. Inspired by the notion from the axiomatic memory model literature, our characterization of correct reorderings is general and is applicable to any libraries equipped with a suitable commutativity specification. The extended-doesn't-commute (WDC) analysis is a standalone contribution of this work and is applicable not only concurrency bug detection but stateless model checking as well.

To combat real world performance challenges of partial order based predictive analysis, we further address the major performance bottleneck of tracking conflicting critical sections and proposed efficient data structures to store the mutex-access metadata on shadow memory. This allows OVPredict to scale to traces with billions of events, which are not uncommon for large applications such as Chromium.

Inspired by the characterization of generalized correct reorderings, we further investigate its application in stateless model checking. Specifically, we propose an SMC algorithm that is able to exploit commutativity specification of library interface. The key challenge in this work is how to systematically enumerate all sound SC-equivalent interleavings (completeness) without redundancy (optimality). Our algorithm borrows the insight from the `EDC` analysis and leverages access point representation to detect conflicts between method invocations in constant time. On SV-COMP benchmarks, our tool NCMC achieves exponential speedup on several benchmarks that use commutative data structures.

The body of work presented in this thesis shows our effort to bridging the gap between the predictive analysis and stateless model checking field. The key insight underlying both OVPredict and NCMC is that dependency can be reasoned at the library method invocation level which encapsulates low-level instructions. We believe that developers can benefit from our tools when developing, testing, and diagnosing concurrent programs. Furthermore, we believe that our work can serve as a baseline for future program analysis and model checking techniques that incorporate dependency reasoning at the library interface level.

REFERENCES

[1] "Chromium issue 841280: heap-use-after-free in blinkgc." https://crbug.com/841280, 2018.

[2] SV-COMP, "Sv-comp." https://sv-comp.sosy-lab.org/2020/, 2019.

[3] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 329–339, ACM, 2008.

[4] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, (New York, NY, USA), 2019.

[5] J. Burnim, K. Sen, and C. Stergiou, "Testing concurrent programs on relaxed memory models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), p. 122–132, Association for Computing Machinery, 2011.

[6] H.-J. Boehm, "How to miscompile programs with "benign" data races," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, (USA), p. 3, USENIX Association, 2011.

[7] J. Huang, "Ufo: Predictive concurrency use-after-free detection," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 609–619, ACM, 2018.

[8] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[9] J. Alglave, L. Maranget, and M. Tautschnig, “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” *ACM Trans. Program. Lang. Syst.*, vol. 36, pp. 7:1–7:74, July 2014.

[10] J. Huang, P. O. Meredith, and G. Rosu, “Maximal sound predictive race detection with control flow abstraction,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), pp. 337–348, ACM, 2014.

[11] J. Huang and A. K. Rajagopalan, “Precise and maximal race detection from incomplete traces,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 462–476, ACM, 2016.

[12] J. Huang and C. Zhang, “Persuasive prediction of concurrency access anomalies,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA ’11)*, (New York, NY, USA), 2011.

[13] Y. Yu, T. Rodeheffer, and W. Chen, “Racetrack: Efficient detection of data race conditions via adaptive tracking,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP ’05)*, (New York, NY, USA), 2005.

[14] K. Sen, “Race directed random testing of concurrent programs,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’08, (New York, NY, USA), pp. 11–21, ACM, 2008.

[15] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, “Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP ’19)*, (New York, NY, USA), 2019.

[16] M. Said, C. Wang, Z. Yang, and K. Sakallah, “Generating data race witnesses by an smt-based analysis,” in *Proceedings of the Third International Conference on NASA*

Formal Methods, NFM'11, (Berlin, Heidelberg), pp. 313–327, Springer-Verlag, 2011.

[17] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, (New York, NY, USA), pp. 387–400, ACM, 2012.

[18] S. Huang and J. Huang, "Speeding up maximal causality reduction with static dependency analysis," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[19] T. F. ŞerbănuŢă, F. Chen, and G. Roşu, "Maximal causal models for sequentially consistent systems," in *Runtime Verification* (S. Qadeer and S. Tasiran, eds.), (Berlin, Heidelberg), pp. 136–150, Springer Berlin Heidelberg, 2013.

[20] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic race prediction in linear time," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), pp. 157–170, ACM, 2017.

[21] J. Roemer, K. Genç, and M. D. Bond, "High-coverage, unbounded sound predictive race detection," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 374–389, ACM, 2018.

[22] K. Genç, J. Roemer, Y. Xu, and M. D. Bond, "Dependence-aware, unbounded sound predictive race detection," *Proc. ACM Program. Lang.*, vol. 3, pp. 179:1–179:30, Oct. 2019.

[23] J. Roemer, K. Genç, and M. D. Bond, "Practical predictive race detection," *CoRR*, vol. abs/1905.00494, 2019.

[24] A. Pavlogiannis, "Fast, sound and effectively complete dynamic race prediction," in *Proceedings of the 47th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '20, ACM, 2020.

[25] U. Mathur, M. S.Bauer, and M. Viswanathan, "Sound dynamic deadlock prediction in linear time." URL: http://umathur3.web.engr.illinois.edu/papers/dcp.pdf.

[26] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 309–318, USENIX, 2012.

[27] J. Edge, "The kernel address sanitizer." `https://lwn.net/Articles/612153/`, 2014.

[28] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[29] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*, (New York, USA), Dec. 2009.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," 1997.

[31] J. Roemer and M. D. Bond, "Online set-based dynamic analysis for sound predictive race detection," *CoRR*, vol. abs/1907.08337, 2019.

[32] Intel Corporation, "Intel Inspector," 2016. `https://software.intel.com/en-us/intel-inspector-xe`.

[33] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, (Berkeley, CA, USA), 2010.

[34] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," *SIGARCH Comput. Archit. News*, vol. 37, p. 25–36, Mar. 2009.

[35] M. Elver, P. E. McKenney, D. Vyukov, A. Konovalov, A. Potapenko, K. Serebryany, A. Stern, A. Parri, A. Yokosawa, P. Zijlstra, W. Deacon, D. Lustig, B. Feng, J. Fernandes, J. Alglave, and L. Maranget, "Kcsan: Concurrency bugs should fear the big bad data-race detector (part 2)." `https://lwn.net/Articles/816854/`, 2020.

[36] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 255–268, USENIX Association, July 2019.

[37] F. Chen, T. F. Serbanuta, and G. Rosu, "jpredictor: A predictive runtime analysis tool for java," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), pp. 221–230, ACM, 2008.

[38] P. Liu, O. Tripp, and X. Zhang, "Ipa: Improving predictive analysis with pointer analysis," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 59–69, ACM, 2016.

[39] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen, "Commutativity race detection," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 305–315, ACM, 2014.

[40] U. Mathur, A. Pavlogiannis, and M. Viswanathan, "Optimal prediction of synchronization-preserving races," 2020.

[41] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, (New York, NY, USA), pp. 174–186, ACM, 1997.

[42] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," *SIGPLAN Not.*, vol. 49, pp. 373–384, Jan. 2014.

[43] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for tso and pso," in *Tools and Algorithms for the Construction*

and Analysis of Systems (C. Baier and C. Tinelli, eds.), (Berlin, Heidelberg), pp. 353–367, Springer Berlin Heidelberg, 2015.

[44] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), pp. 250–259, ACM, 2015.

[45] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, "Unfolding-based Partial Order Reduction," in *26th International Conference on Concurrency Theory (CONCUR 2015)* (L. Aceto and D. de Frutos Escrig, eds.), vol. 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 456–469, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[46] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), pp. 165–174, ACM, 2015.

[47] S. Huang and J. Huang, "Maximal causality reduction for tso and pso," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 447–461, ACM, 2016.

[48] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for c/c++ concurrency," *Proc. ACM Program. Lang.*, vol. 2, pp. 17:1–17:32, Dec. 2017.

[49] E. Albert, M. Gómez-Zamalloa, M. Isabel, and A. Rubio, "Constrained dynamic partial order reduction," in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 392–410, Springer International Publishing, 2018.

[50] T. Gehr, D. Dimitrov, and M. Vechev, "Learning commutativity specifications," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 307–

323, Springer International Publishing, 2015.

[51] K. Bansal, E. Koskinen, and O. Tripp, "Automatic generation of precise and useful commutativity conditions," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer and M. Huisman, eds.), (Cham), pp. 115–132, Springer International Publishing, 2018.

[52] F. Mattern, "Virtual time and global states of distributed systems," in *PARALLEL AND DISTRIBUTED ALGORITHMS*, pp. 215–226, North-Holland, 1988.

[53] A. Raad, M. Doko, L. Rožić, O. Lahav, and V. Vafeiadis, "On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models," *Proc. ACM Program. Lang.*, vol. 3, pp. 68:1–68:31, Jan. 2019.

[54] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 121–133, ACM, 2009.

[55] "Machine types | compute engine documentation." URL: https://cloud.google.com/compute/docs/machine-types.

[56] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 325–336, ACM, 2009.

[57] "Chromium issue 904714: heap-use-after-free on sw::renderer::finishrendering." `https://crbug.com/904714`, 2018.

[58] "Chromium issue 944424: Uaf in taskqueueimpl::createtaskrunner." `https://crbug.com/944424`, 2019.

[59] "Chromium issue 945370: Uaf in indexeddb." `https://crbug.com/945370`, 2019.

[60] "A collection of concurrency bugs on github." `https://github.com/jieyu/concurrency-bugs`.

[61] "Chromium issue tracker." `https://crbug.com`.

[62] D. Rhodes, C. Flanagan, and S. N. Freund, "Bigfoot: Static check placement for dynamic race detection," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), p. 141–156, Association for Computing Machinery, 2017.

[63] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), pp. 96–110, ACM, 2019.

[64] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, July 1990.

[65] D. Kim and M. C. Rinard, "Verification of semantic commutativity conditions and inverse operations on linked data structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 528–541, ACM, 2011.

[66] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali, "Exploiting the commutativity lattice," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 542–555, ACM, 2011.

[67] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson, "Stateless model checking for power," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 134–156, Springer International Publishing, 2016.

[68] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, “Optimal stateless model checking under the release-acquire semantics,” *Proc. ACM Program. Lang.*, vol. 2, pp. 135:1–135:29, Oct. 2018.

[69] S. Lu, J. Tucek, F. Qin, and Y. Zhou, “Avio: Detecting atomicity violations via access interleaving invariants,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’06)*, (New York, NY, USA), 2006.

[70] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, “Atom-aid: Detecting and surviving atomicity violations,” *SIGARCH Comput. Archit. News*, vol. 36, p. 277–288, June 2008.

[71] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, “2ndstrike: Toward manifesting hidden concurrency typestate bugs,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11)*, (New York, NY, USA), 2011.

[72] W. Zhang, C. Sun, and S. Lu, “Conmem: Detecting severe concurrency bugs through an effect-oriented approach,” *SIGPLAN Not.*, vol. 45, p. 179–192, Mar. 2010.

[73] B. Lucia and L. Ceze, “Finding concurrency bugs with context-aware communication graphs,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 553–563, 2009.

[74] L. Chew and D. Lie, “Kivati: Fast detection and prevention of atomicity violations,” in *Proceedings of the 5th European Conference on Computer Systems (EuroSys ’10)*, (New York, NY, USA), 2010.

[75] B. Norris and B. Demsky, “Cdschecker: Checking concurrent data structures written with c/c++ atomics,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA ’13, (New York, NY, USA), pp. 131–150, ACM, 2013.

[76] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 211–222, ACM, 2007.

[77] E. Koskinen, M. Parkinson, and M. Herlihy, "Coarse-grained transactions," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, (New York, NY, USA), pp. 19–30, ACM, 2010.

[78] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), pp. 207–216, ACM, 2008.

[79] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis framework for parallelizing compilers," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, (New York, NY, USA), pp. 54–67, ACM, 1996.

[80] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), pp. 2–15, ACM, 2009.

[81] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, "Testing atomicity of composed concurrent operations," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, (New York, NY, USA), pp. 51–64, ACM, 2011.

[82] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," 02 2015.

[83] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable Use-after-free Detection," in *EuroSys*, Apr. 2017.

[84] Y. Younan, “Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers,” 01 2015.

[85] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, “A randomized scheduler with probabilistic guarantees of finding bugs,” in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 167–178, ACM, 2010.

[86] M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, (New York, NY, USA), pp. 446–455, ACM, 2007.

[87] Y. Cai and L. Cao, “Effective and precise dynamic detection of hidden races for java programs,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 450–461, ACM, 2015.

[88] M. Eslamimehr and J. Palsberg, “Race directed scheduling of concurrent programs,” in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’14, (New York, NY, USA), pp. 301–314, ACM, 2014.

[89] T. A. Henzinger, R. Jhala, and R. Majumdar, “Race checking by context inference,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI ’04, (New York, NY, USA), pp. 1–13, ACM, 2004.

[90] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, “Precise static happens-before analysis for detecting uaf order violations in android,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, p. 276–287, IEEE, Apr 2019.

[91] A. Singh, R. Pai, D. D’Souza, and M. D’Souza, *Static Analysis for Detecting High-Level Races in RTOS Kernels*, vol. 11800, p. 337–353. Springer International Publishing, 2019.