Runtime Verification and Debugging of Concurrent Software

Lu Zhang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Chao Wang, Chair
Michael S. Hsiao, Co-Chair
Haibo Zeng
Changhee Jung
Dongyoon Lee

June 10, 2016
Blacksburg, Virginia

Keywords: Concurrency, Verification, Debugging, Program Repair, Quasi Linearizability,
Concurrent Data Structure, Web Application, JavaScript

Runtime Verification and Debugging of Concurrent Software

Lu Zhang

(ABSTRACT)

Our reliance on software has been growing fast over the past decades as the pervasive use of computer and software penetrated not only our daily life but also many critical applications. As the computational power of multi-core processors and other parallel hardware keeps increasing, concurrent software that exploit these parallel computing hardware become crucial for achieving high performance. However, developing correct and efficient concurrent software is a difficult task for programmers due to the inherent nondeterminism in their executions. As a result, concurrency related software bugs are among the most troublesome in practice and have caused severe problems in recent years.

In this dissertation, I propose a series of new and fully automated methods for verifying and debugging concurrent software. They cover the detection, prevention, classification, and repair of some important types of bugs in the implementation of concurrent data structures and client-side web applications. These methods can be adopted at various stages of the software development life cycle, to help programmers write concurrent software correctly as well as efficiently.

Acknowledgment

I would like to express my sincere gratitude and appreciation to my advisor, Dr. Chao Wang, for his guidance, patience, and support — this dissertation would not have been possible without him. His advice has been priceless not only for my research but also for my career.

I would like to thank Dr. Michael S. Hsiao, Dr. Haibo Zeng, Dr. Changhee Jung and Dr. Dongyoon Lee for kindly serving on my dissertation committee. They have been helpful every step of the way.

I want to thank my parents, Liangcun Zhang and Zunjuan Fan, for their love and support all these years. Finally, I want to thank my girlfriend, Dandan Zhang, for always being there for me.

Lu Zhang

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Detecting and repairing software defects or *bugs* are the most expensive part of the software development process, typically taking up more than 50% of the total development cost [14]. According to a widely cited federal study [42], software bugs are costing the U.S. economy an estimated $59.5 billion annually, or 0.6% of the GDP. However, despite intensive research in the past decades, software developers today still do not have the support of adequate tools for detecting and fixing software bugs. For example, in practice, fixing one software bug can take more than a month [70, 21], and many known bugs are never fixed due to the lack of human resources. More alarmingly, approximately 70% of the bug patches are incorrect in their first releases [117], thus leading to more product delays [100], financial losses [90], and security vulnerabilities.

The situation is exacerbated by the widespread use of multi-core processors in all application domains, because the current software development tools do not help ordinary programmers write reliable and efficient concurrent software. As a matter of fact, concurrency related bugs are among the most troublesome software bugs in practice. First, they are hard to detect because of the thread-scheduling nondeterminism, multiple runs of the same program may exhibit different behaviors even under the same program input. Therefore, bugs may manifest themselves in some executions but not others. Second, the total number of possible

interleavings of the concurrent threads can be astronomically large, which makes exhaustively testing the program difficult or even impossible. Third, concurrency bugs are hard to avoid during software development because programmers often think sequentially [116] and thus may miss the corner case interleavings. Finally, even after a concurrency bug is detected, diagnosing the failure is difficult because it is not easy for programmers to replay the erroneous interleaving.

Existing software development tools are inadequate for verifying and debugging concurrent software because they either generate too many false alarms, or may miss real errors, or lack sufficient information to aid programmers in bug diagnosis and repair. In practice, concurrent software debugging often involves the programmers simply staring at the source code [82], which is neither economical nor reliable. For example, a developer-provided bug patch in the `Mozilla` project was incorrect [65], leading to new bugs in the field whose subsequent patches led to even more bugs, until they were fixed more than a year later. Thus, there is an urgent need to develop new and automated techniques for the detection, prevention, diagnosis, and repair of concurrency bugs.

1.1 Background

In concurrent software, multiple operations may be executed simultaneously to improve the overall performance. However, the inherently nondeterministic nature of such execution also brings challenges to testing and verification. Specifically, incorrect use of shared resources can lead to race conditions, deadlocks and performance bugs. Such concurrency bugs exist in not only multithreaded applications but also the emerging web and mobile applications built upon event-driven programming frameworks.

Detecting Concurrency Bugs There is a large body of work on detecting bugs in concurrent software. Some of these existing methods focus on data-races [111, 163, 108, 101,

52, 62, 113] while the others focus on deadlocks [84, 97, 5, 103] and atomicity violations [40, 36, 119, 132, 63, 53, 64, 115, 54]. Although detecting such generic and low-level concurrency bugs are useful, the main problem of these existing methods is that they cannot detect high-level concurrency bugs, such as violations of the programmer's intent regarding the functionality. Therefore, in this dissertation, I focus on developing new methods for detecting high-level concurrency bugs. Specifically, I focus on detecting violations of a high-level correctness condition, called *(quasi) linearizability*, in the implementation of concurrent data structures.

Preventing Concurrency Bugs There is also a large body of work on avoiding failures caused by concurrency bugs at run time. For example, Yu and Narayansamy [158] collect event ordering constraints from good profiling runs and use them to restrict the thread interleaving in production runs; therefore, they can avoid previously untested (and potentially buggy) executions. Similar techniques are also proposed for avoiding data-races [104, 107], atomicity violations [85, 77, 145, 58, 78, 143], and deadlocks [61, 141, 142, 150]. However, these existing methods focus primarily on the monitoring and control of memory read/write instructions and thus have extremely high runtime overhead. In contrast, I focus on controlling the order of method calls, instead of the order of read/write instructions, to significantly reduce the runtime overhead of the failure avoidance techniques.

Classifying Concurrency Bugs Although a large number of concurrency bugs are caused by race conditions, not all race conditions are real bugs. In fact, the vast majority of race-condition warnings reported by existing tools are either bogus, which means they can never appear in the actual program execution, or benign, which means they do not lead to observable differences in the program state. However, automatically identifying the harmful race conditions is a challenging task. Narayanasamy et al. [96] proposed perhaps the first deterministic replay-based method for separating harmful data-races from the harmless ones. During the execution of the program, they employ a checkpointing tool to take snapshots

of the main memory, and then compare the snapshots to decide if the data-race can lead to different program states. Similar techniques are also used by race-directed testing tools such as RaceFuzzer [114], Portend [66], CTrigger [98], PENELOPE [123], and Maple [159]. However, these existing methods focus primarily on data-races in multithreaded C++/Java applications. In contrast, I focus on classifying race conditions in JavaScript-based, client-side web applications.

Repairing Concurrency Bugs

Broadly speaking, program repair is a special case of program synthesis [35, 88, 102, 105, 106], which aims to generate correct implementations of formal specifications automatically. For repairing concurrency bugs, in particular, a common strategy is adding or removing synchronization operations to the program code to eliminate the erroneous thread interleavings [130, 71, 26, 25, 156]. Indeed, a large number of tools have been developed for automatically repairing concurrency bugs in multithreaded C/C++/Java applications [43, 59, 109, 67]. However, none of these existing tools can handle concurrency bugs in event-driven programs. Although the ARROW tool developed by Wang et al. [140] can automatically patch synchronization bugs in client-side web applications, it relies on purely static analysis techniques and thus cannot robustly handle web applications that rely on the use of dynamic features of JavaScript and third-party libraries.

1.2 Dissertation Contributions

The objective of my research is to develop new and automated methods for detecting, preventing, classifying and repairing concurrency bugs. Toward this end, I aim to address the following technical challenges:

- How to effectively detect concurrency bugs?

- How to mitigate concurrency bugs at run time to avoid program crashes?
- How to automatically diagnose concurrency bugs?
- How to repair concurrency bugs?

Accordingly, this dissertation makes four main contributions. The first two are related to the detection and avoidance of concurrency bugs in multithreaded software, while the last two are related to the classification and automated repair of bugs in event-driven web applications.

Contribution 1: Detecting Quasi Linearizability Violations in Concurrent Data Structures I propose a new method for runtime checking of a relaxed consistency property called *quasi linearizability* for concurrent data structures. Quasi linearizability generalizes the standard notion of *linearizability* by introducing additional nondeterminism into the parallel computations quantitatively and then exploiting such nondeterminism to improve the runtime performance. However, ensuring the quantitative aspects of this correctness condition in the low-level code of the concurrent data structure implementation is a difficult task. The new runtime verification method is the first fully automated method for checking quasi linearizability in the C/C++ code of concurrent data structures. It guarantees that all the reported quasi linearizability violations manifested by the concurrent executions are real violations. I have implemented the new method in a software tool based on the LLVM compiler and a systematic concurrency testing tool called Inspect. The experimental evaluation shows that the new method is effective in detecting quasi linearizability violations in the source code implementations of concurrent data structures.

Contribution 2: Preventing Type-State Violations in Multithreaded Applications I propose a new method for runtime prevention of type-state violations in multithreaded applications due to erroneous thread interleavings. The new method employs a combination of static and dynamic program analysis techniques to control the execution order of the method calls to suppress illegal call sequences. The legal behavior of a shared object is

specified by a type-state automaton, which serves as the guidance for our method to delay certain method calls at run time. My main contribution is a new theoretical framework for ensuring that the runtime prevention strategy is always safe, i.e., they do not introduce new erroneous interleavings. Furthermore, whenever the static program analysis is precise enough, our method guarantees to steer the program to a failure-free interleaving as long as such interleaving exists. I have implemented the new method in a tool based on the LLVM compiler framework. The experiments on a set of multithreaded C/C++ applications show that the method is both efficient and effective in suppressing concurrency related type-state violations.

Contribution 3: Classifying Race Conditions in Web Applications I propose a replay-based method for classifying race-condition warnings in web applications, by re-executing each pair of racing events in two different orders and comparing the program states. Race conditions are common in modern web applications based on JavaScript. Unfortunately, existing tools for detecting race conditions report many warnings, of which only a small portion are real, i.e., they actually change the program states and therefore demand the attention of developers. Since manually classifying race-condition warnings is labor-intensive and error-prone, directly reporting them to developers is counter-productive. Automation promises to remove the burden of filtering bogus warnings from developers and allow them to focus on the more important bugs. I have implemented our method and evaluated it on a large set of real websites from Fortune-500 companies. The experiments show that the new method is both effective in filtering bogus warnings and scalable for practical use.

Contribution 4: Repairing Race Conditions in Web Applications

I propose a dynamic analysis-based tool, named *RCrepair*, for automatically repairing race conditions in web applications. *RCrepair* takes the HTML file and the race condition as input and returns the modified HTML file as output. The output will be presented to developers,

who ultimately decide whether the suggested repair is appropriate. Automation brought by *RCrepair* can significantly ease the burden on developers, allowing them to diagnose and repair bugs quicker. I have implemented *RCrepair* and evaluated it on a large set of real websites of the Fortune-500 companies. The experimental results show the new method is both effective in computing repairs and efficient for practical use.

1.3 Organization

The remainder of this dissertation is organized as follows.

In Chapter 2, I present a new method for detecting standard and quasi linearizability violations in the C/C++ code that implement concurrent data structures.

In Chapter 3, I present a new method that leverages a combination of static and dynamic analysis techniques to avoid type-state violations in multithreaded C++ applications.

In Chapter 4, I present a new method for automatically classifying harmful race-condition warnings in JavaScript-based web applications.

In Chapter 5, I present a new method for leveraging dynamic analysis to automatically repair race conditions in JavaScript-based web applications.

Finally, in Chapter 6, I give my conclusions and outline the future work.

Chapter 2

Detecting Quasi Linearizability Violations

Concurrent data structures are the foundation of many multi-core and high-performance software systems. By providing a cost-effective way to reduce the memory contention and increase the scalability, they have found many applications ranging from embedded computing to distributed systems such as the cloud. However, implementing concurrent data structures is not an easy task due to the subtle interactions of low-level concurrent operations and the often astronomically many thread interleavings. In practice, even a few hundred lines of highly concurrent C/C++ code can pose severe challenges for testing and debugging.

Linearizability [50, 49] is the *de facto* correctness condition for implementing concurrent data structures. It requires that every interleaved execution of the methods of a concurrent object to be equivalent, in some sense, to a sequential execution of these methods. This is extremely useful as a correctness condition for application developers because, as long as the program is correct while running in the sequential mode using the standard (sequential) data structure, switching to a concurrent version of the same data structure would not change the program behavior. Although being linearizable alone does not guarantee correctness of the program, not satisfying the linearizability requirement almost always indicates that the

implementation is buggy.

Quasi linearizability [4] is a quantitative relaxation of linearizability that has attracted a lot of attention in recent years [99, 69, 68, 48, 47]. For many highly parallel applications, the standard notion of linearizability often imposes unnecessary restrictions on the implementation, thereby leading to severe performance bottlenecks. Quasi linearizability has the advantage of preserving the intuition of standard linearizability while providing some additional flexibility in the implementation. For example, the task queue used in the scheduler of a thread pool does not need to follow the strict FIFO order. That is, one can use a relaxed queue that allows some tasks to be overtaken occasionally if such relaxation leads to superior runtime performance. Similarly, concurrent data structures used for web cache need not follow the strict semantics of the standard versions since occasionally getting the stale data is acceptable. In distributed systems, a unique identifier (id) generator does not need to be a perfect counter because to avoid becoming a performance bottleneck; it is often acceptable for the ids to be out of order occasionally, as long as it happens within a bounded time frame. Quasi linearizability allows the concurrent objects to have such occasional deviations from the standard semantics in exchange for higher performance.

While quasi linearizable concurrent data structures can have tremendous runtime performance advantages, ensuring the quantitative aspects of this correctness condition in the actual implementation is not an easy task. In this chapter, we propose the first fully automated runtime verification method, called *Round-Up*, for checking *quasi* linearizability violations of concurrent data structures. To the best of our knowledge, prior to *Round-Up*, there does not exist any method for checking, for example, the `deq` operation of a relaxed queue is not overtaken by other `deq` operations for more than k times. Most of the existing concurrency bug detection methods focus on detecting simple bug patterns such as deadlocks, data-races, and atomicity violations, as opposed to quantitative properties manifested in *quasi* linearizability.

Our work also differs from the large body of work on checking standard linearizability, which

is not a quantitative property. Broadly speaking, existing methods for checking *standard* linearizability fall into three groups. The first group consists of methods based on constructing mechanical proofs [127, 128], which typically require significant user intervention. The second group consists of automated methods based on techniques such as model checking [79, 129, 19], which work only on finite-state models or abstractions of the actual concurrent data structures. The third group consists of runtime verification tools that can directly check the implementation at the source-code level, but only for standard linearizability.

In contrast, *Round-Up* is the first runtime verification method for checking quasi linearizability in the source code implementations of concurrent data structures. It is fully automated in that the method does not require the user to provide functional specifications or to annotate the linearization points in the code. It takes the source code of a concurrent object o, a test program P that uses o, and a quasi factor K as input, and returns either true or false as output. It also guarantees to report only real linearizability violations.

We have implemented the method in a software tool based on the LLVM compiler [3] and a systematic concurrency testing tool called Inspect [153, 154]. Our method can handle C/C++ programs that are written using the POSIX threads and GNU built-in atomic functions. Our experimental evaluation of the tool on a large set of concurrent data structure implementations shows that the new method is effective in detecting both standard and quasi linearizability violations. For example, we have found several real implementation bugs in the *Scal* suite [112], which is an open-source package that implements some recently published concurrent data structures. The bugs that we found in the *Scal* benchmarks have been confirmed by the developers.

To sum up, this chapter makes the following contributions:

- We propose the first method for runtime checking of quasi linearizability in the source-code implementation of low-level concurrent data structures. The new method is fully automated and can guarantee that all the reported quasi linearizability violations are real violations.

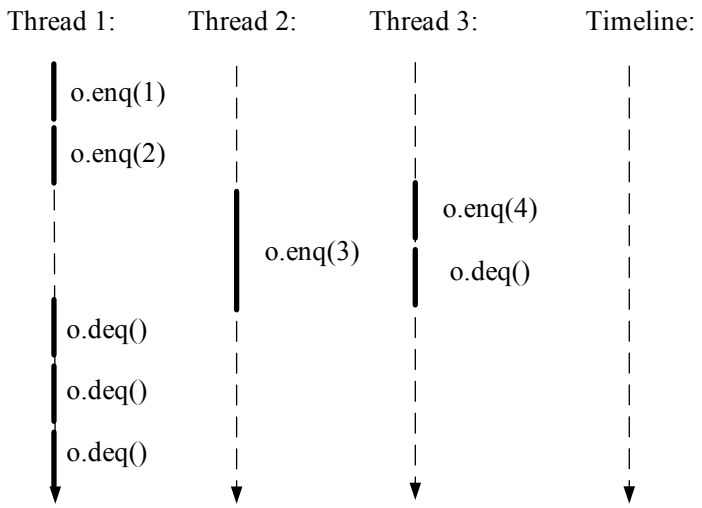


Figure 2.1: A 3-threaded program that uses object *o*. Thread 1 starts by adding values 1 and 2 to the queue before creating two child threads. Then it waits for the child threads to terminate before removing another three data items. Here `enq(3)` runs concurrently with `enq(4)` and `deq()` in Thread 3.

- We have implemented the new method in a software tool based on LLVM and Inspect and evaluated the tool on the real C/C++ code of a large set of concurrent data structures. The results demonstrate that the new method is effective in detecting standard and quasi linearizability violations.

2.1 Motivating Examples

In this section, we illustrate the standard and quasi linearizability properties and outline the technical challenges in checking such properties. Figure 2.1 shows a multithreaded program that invokes the `enq` and `deq` methods of a queue, where `enq(i)` adds data item `i` to the end of the queue and `deq` removes the data item from the head. If Thread 2 executes `enq(3)` atomically, i.e., without interference from Thread 3, there will be three interleaved executions of the methods, all of which behave like a single-threaded execution. The sequential histories, shown in Figure 2.2, satisfy the standard FIFO semantics of the queue. Therefore, we call them *legal sequential histories*.

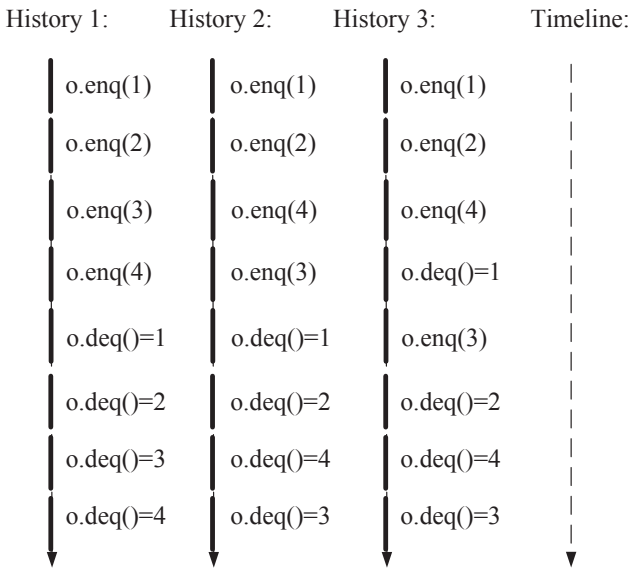


Figure 2.2: The set of *legal sequential histories* generated by the program in Figure 2.1. These legal sequential histories form the *sequential specification*.

If the time interval of `enq(3)`, which starts at the method's invocation and ends at its response, overlaps with the time intervals of `enq(4)` and `deq()`, the execution is no longer sequential. In this case, the interleaved execution is called a *concurrent history*. When the implementation of the queue is linearizable, no matter how the instructions of `enq(3)` interleave with the instructions of `enq(4)` and `deq()`, the external behavior of the queue would remain the same. We say that the queue is *linearizable* if the sequence of `deq` values of any *concurrent history* matches one of the three legal sequential histories in Figure 2.2. On the other hand, if the sequence of `deq` values is 3,2,1,4 in a concurrent history, for example, we say that the concurrent history has a linearizability violation because the object no longer behaves like a FIFO queue.

However, being linearizable often means that the implementation has significant performance overhead, for example, when it is used by a large number of concurrent threads. For a quasi linearizable queue, in contrast, it is acceptable to have the `deq` values being out of order occasionally, if such relaxation of the standard FIFO semantics can help improve the

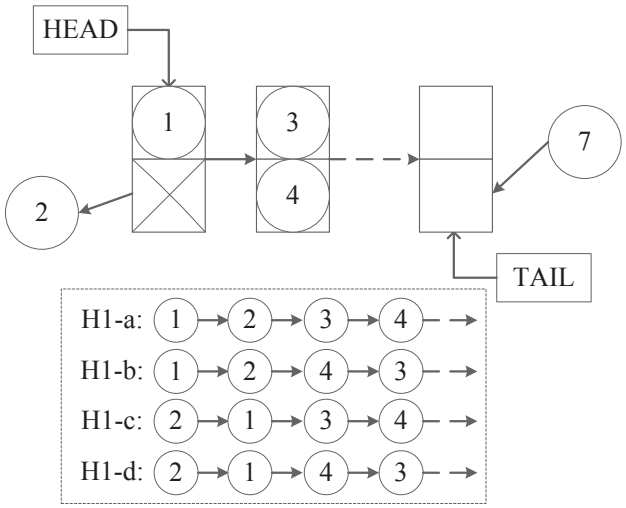


Figure 2.3: An example implementation of *1-quasi linearizable* queue, where each of the linked list item is a segment that holds two data items. The first `deq` randomly returns a value from the set $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the set $\{3, 4\}$ and the fourth `deq` returns the remaining one.

performance. For example, instead of using a standard linked list to implement the queue, one may use a linked list of 2-cell segments to implement the 1-quasi linearizable queue (Figure 2.3). In this case, the `deq` operation may remove any of the two data items in the head segment. By using randomization, it is possible for two threads to remove different data items from the head simultaneously without introducing memory contention.

Assume that the relaxed queue contains four values 1,2,3,4 initially. The first two `deq` operations would retrieve either 1,2 or 2,1, and the next two `deq` operations would retrieve either 3,4 or 4,3. Together, there are four possible combinations as shown in Figure 2.3. Among them, $H1$-a is linearizable. The other three are not linearizable, but they are considered as *1-quasi linearizable*, meaning that `deq` values in these concurrent histories are out-of-order by at most one step.

However, implementing such quasi linearizable concurrent data structures is a difficult task. Subtle bugs can be introduced during both the design phase and the implementation phase. Consider an alternative way of implementing the 1-quasi linearizable queue as illustrated

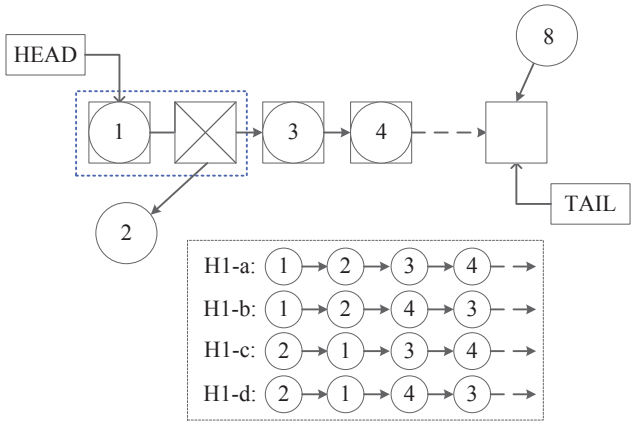


Figure 2.4: An alternative implementation of *1-quasi linearizable* queue, which is based on the random-dequeued queue. The first `deq` randomly returns a value from $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the new window $\{3, 4\}$ and the fourth `deq` returns the remaining one.

in Figure 2.4, where the first two data items are grouped into a virtual window. A `deq` operation may retrieve any of the first 2 data items from the head based on randomization. Furthermore, only after both data items in the current window are removed, will the `deq` operation move on to retrieve data items in the next window. The resulting behavior of this implementation should be identical to that of the segmented queue.

However, a subtle bug would appear if one ignores the use of the *virtual window*. For example, if `deq` always returns one of the first two data items in the current queue, although it appears to be correct, the implementation would not be considered as 1-quasi linearizable. In this case, it is possible for some data item to be over-taken indefinitely, thereby making the data structure unsuitable for applications where a 1-quasi queue is desired. For example, if every time the `deq` operation removes *the second data item in the list*, we would get a sequence of `deq` values as follows: 2,3,4,..., where value 1 is left in the queue indefinitely.

The above example demonstrates the need for a new verification method that can help detect violations of the quantitative properties in *quasi* linearizable concurrent data structures. Unfortunately, existing concurrency bug checking tools focus primarily on simple bug pat-

terns such as deadlocks and data-races. They are not well suited for checking quantitative properties in the low-level code that implements concurrent data structures. To the best of our knowledge, the method proposed in this chapter is the first runtime verification method for detecting quasi linearizability violations in the source code of concurrent data structures.

2.2 Preliminaries

2.2.1 Linearizability

We follow the notation in [50, 49] to define a *history* as a sequence of events, denoted $h = e_1 e_2 \dots e_n$, where each event is either a method invocation or a method response for an object. When there are multiple objects involved in the history h, we use $\rho = h|o$ to denote the projection of h to the object o, which results in a subsequence of events related only to this object. When there are multiple threads, let $\rho|T$ denote the projection of history ρ to thread T, which is the subsequence of events of this thread. Two histories ρ and ρ' are equivalent, denoted $\rho \sim \rho'$, if and only if $\rho|T_i = \rho'|T_i$ for all thread T_i, where $i = 1, \dots, k$. Therefore, two equivalent histories have the same set of events, but the events may be arranged in different orders.

A *sequential history* is one that starts with a method invocation, and each method invocation is followed immediately by the matching response; in other words, no two method call intervals are overlapping. Otherwise, the history is called a *concurrent history*. Let $<_\rho$ be the precedence relation of events in the history ρ. Let $\rho[e]$ be the index of the event e in ρ. For two events e_1 and e_2, we say that $e_1 <_\rho e_2$ if and only if $\rho[e_1] < \rho[e_2]$.

Definition 1 *A* sequentialization *of a concurrent history ρ is a sequential history ρ' such that (1) $\rho' \sim \rho$, meaning that they share the same set of events, and (2) $\forall e_i, e_j :\ (e_i <_\rho e_j)$ implies $(e_i <_{\rho'} e_j)$. That is, the non-overlapping method calls in ρ retain their execution*

order in ρ', whereas the overlapping method calls in ρ may take effect in any order in ρ'.

A *sequential specification* of object o, denoted $spec(o)$, is the set of all *legal* sequential histories – they are histories that conform to the semantics of the object. For example, a legal sequential history of a queue is one in which all the `enq/deq` values follow the FIFO order.

Definition 2 *A concurrent history ρ is linearizable with respect to a sequential specification $spec(o)$ if and only if it has a sequentialization ρ' such that $\rho' \in spec(o)$. In other words, as long as the concurrent history ρ can be mapped to at least one legal sequential history $\rho' \in spec(o)$, it is considered as linearizable.*

2.2.2 Quasi Linearizability

The notion of quasi linearizability relies on the permutation distance between two sequential histories. Let $\rho' = e'_1 e'_2 \dots e'_n$ be a permutation of $\rho = e_1 e_2 \dots e_n$. Let $\Delta(\rho, \rho')$ be the distance between ρ and ρ' defined as $max_{e \in \rho}\{\ |\rho[e] - \rho'[e]|\ \}$. We use $\rho[e]$ and $\rho'[e]$ to denote the index of event e in ρ and ρ', respectively. Therefore, $\Delta(\ \rho,\ \rho'\ )$ is the maximum distance that some event in ρ has to travel to its new position in ρ'.

Quasi linearizability is often defined on a subset of the object's methods. Let $Domain(o)$ be the set of all operations of object o. Let $d \subset Domain(o)$ be a subset. Let $Powerset(Domain(o))$ be the set of all subsets of $Domain(o)$. Let $D \subset Powerset(Domain(o))$ be a subset of the powerset.

Definition 3 *The* quasi-linearization factor *(or quasi factor) for a concurrent object o is a function $Q_o : D \to N$, where $D \subset Powerset(Domain(o))$ and N is the set of natural numbers.*

For example, a queue where `enq` operations always follow the FIFO order, but `deq` values

may be out-of-order by at most K steps, can be specified as follows:

$$D_{\texttt{enq}} = \{\ \langle \texttt{o.enq}(x), \mathit{void}\rangle \mid x \in X\ \}$$
$$D_{\texttt{deq}} = \{\ \langle \texttt{o.deq}(), x\rangle \mid x \in X\ \}$$
$$Q_{\texttt{queue}}(D_{\texttt{enq}}) = 0$$
$$Q_{\texttt{queue}}(D_{\texttt{deq}}) = K$$

Here, $\langle \texttt{o.enq}(x), \mathit{void}\rangle$ represents the invocation of $\texttt{o.enq}(x)$ and the return value *void*, where $x \in X$ is the data value added to the queue. Similarly, $\langle \texttt{o.deq}(), x\rangle$ represents the invocation of $\texttt{o.deq}()$ and the return value x. $Q_{\texttt{queue}}(D_{\texttt{enq}}) = 0$ means that the `enq` events follow the standard FIFO order. $Q_{\texttt{queue}}(D_{\texttt{deq}}) = K$ means that the `deq` events are allowed to be out-of-order by at most K steps. Such relaxed queue can be useful in producer-consumer applications, for example, when data are enqueued by a single producer and dequeued by multiple consumers. By relaxing the ordering of the dequeue operations alone, we allow at most K consumers to retrieve data from the queue simultaneously, without any memory contention. The relaxed semantics allows more freedom in the implementation of the concurrent queue, thereby leading to significant performance improvement [99, 47].

Definition 4 *A concurrent history ρ is* quasi linearizable [4] *with respect to a sequential specification spec(o) and quasi factor Q_o iff ρ has a sequentialization ρ' such that,*

- *either $\rho' \in spec(o)$, meaning that ρ is linearizable and hence is also quasi linearizable, or*
- *there exists a permutation ρ'' of the sequentialization ρ' such that*
  - *$\rho'' \in spec(o)$; and*
  - *$\Delta(\ \rho'|d,\ \rho''|d\ ) \leq Q_o(d)$ for all subset $d \in D$.*

In other words, ρ' needs to be a legal sequential history by itself or be within a bounded distance from a legal sequential history ρ''.

From now on, given a sequential history ρ', we call $\Psi = \{\ \rho'' \mid \Delta(\ \rho'|d,\ \rho''|d\ ) \leq Q_o(d)$ for all $d \in D\ \}$ the set of *quasi-permutations* of ρ'.

Quasi linearizability is compositional in that a history h is quasi linearizable if and only if subhistory $h|o$, for each object o, is quasi linearizable. This allows us to check quasi linearizability on each individual object in isolation, which reduces the computational overhead. Furthermore, the standard notion of linearizability is subsumed by quasi linearizability with $Q_o : D \rightarrow 0$.

2.2.3 Checking (Quasi) Linearizability

There are at least three levels where one can check the (quasi) linearizability property.

- **L1:** check if a concurrent history ρ is linearizable:

$$\exists \text{ sequentialization } \rho' \text{ of history } \rho\text{: } \rho' \in \mathit{spec}(o).$$

- **L2:** check if a concurrent program P is linearizable:

$$\forall \text{ concurrent history } \rho \text{ of } P\text{: } \rho \text{ is linearizable.}$$

- **L3:** check if a concurrent object o is linearizable:

$$\forall \text{ test program } P \text{ that uses object } o\text{: } P \text{ is linearizable.}$$

L3 may be regarded as the full-fledged verification of the concurrent object, whereas L1 and L2 may be regarded as runtime bug detection. In this chapter, we focus primarily on the L1 and L2 checks. That is, given a test program P that uses the concurrent object o, we systematically generate the set of concurrent histories of P and then check if all of these concurrent histories are (quasi) linearizable. Our main contribution is to propose a new algorithm for deciding whether a concurrent history ρ is quasi linearizable.

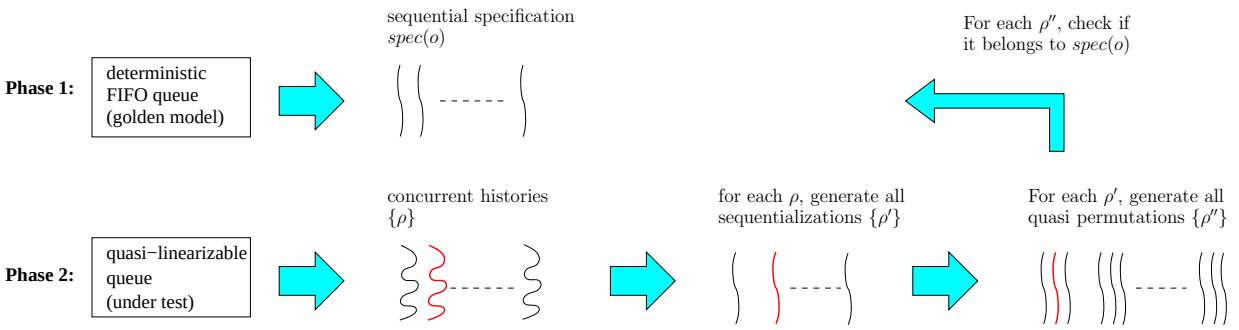


Figure 2.5: The overall flow of our new quasi linearizability checking algorithm.

2.3 Algorithm

The overall algorithm for checking quasi linearizability consists of two phases (see Figure 2.5). In Phase 1, we systematically execute the test program P together with a standard data structure implementation to construct a sequential specification $spec(o)$, which consists of all the legal sequential histories. In Phase 2, we systematically execute the test program P together with the concurrent data structure implementation, and for each concurrent history ρ, check whether ρ is quasi linearizable.

For widely used data structures such as queues, stacks, and priority queues, a sequential version may serve as the golden model in Phase 1. Alternatively, the user may use a specifically configured concurrent data structure as the golden model, e.g., by setting the quasi factor of a relaxed queue implementation to 0, which effectively turns it into a normal queue.

In Phase 1, we use a systematic concurrency testing tool called Inspect [153, 154] to compute all the legal sequential histories. Given a multithreaded C/C++ program and fixed data input, Inspect can be used to systematically generate all possible thread interleavings of the program under the data input. In order to use Inspect in Phase 1, we have modified Inspect to automatically wrap up every method call of a shared object o in a lock/unlock pair. For example, method call `o.enq()` becomes `lock(lk);o.enq();unlock(lk)`, where we assign a lock lk to each object o to ensure that context switches happen only at the method call boundary. In other words, all method calls of object o are executed serially. Furthermore,

Inspect can guarantee that all the possible sequential histories of this form are generated. We leverage these *legal sequential histories* to construct the sequential specification $spec(o)$ of the object o. For any given test program P, the sequential specification $spec(o)$ is represented as the set of all legal sequential histories of the test program P. We focus on checking the linearizability of each individual concurrent object without loss of generality, because linearizability is compositional in that an execution history is linearizable with respect to multiple objects if it is linearizable with respect to each individual object.

In Phase 2, we use Inspect again to compute the set of concurrent histories of the same test program. However, this time, we allow the instructions within the method bodies to interleave freely. This can be accomplished by invoking Inspect in its default mode, without adding the aforementioned lock/unlock pairs. In addition to handling the POSIX thread functions such as mutex lock/unlock and signal/wait, we have extended Inspect to support the set of GNU built-in functions for atomic memory access, which are frequently used in practice for implementing concurrent data structures. Since we use LLVM as the front-end for code instrumentation, it means that we treat LLVM atomic instructions such as `cmpxchg` and `atomicrmw` similar to shared variable write instructions. Here, `cmpxchg` refers to the atomic compare-and-exchange instruction, and `atomicrmw` refers to the atomic read-modify-write instruction. During Phase 2, Inspect will interleave them systematically while generating the concurrent histories.

Our core algorithm for checking whether a concurrent history ρ is quasi linearizable is invoked in Phase 2.

- For each concurrent history ρ, we compute the set Φ of *sequentializations* of ρ (see Definition 1).
- If any $\rho' \in \Phi$ matches a legal sequential history in $spec(o)$, we conclude that ρ is linearizable and therefore is also quasi linearizable.
- Otherwise, for each sequentialization $\rho' \in \Phi$, we compute the set Ψ of *quasi-permutations* of ρ' with respect to the given quasi factor, which defines the distance between ρ' and

each $\rho'' \in \Psi$ (see Definition 4):

- If there exists a quasi permutation ρ'' of ρ' such that $\rho'' \in \mathit{spec}(o)$, then ρ is quasi linearizable.
- Otherwise, ρ is not quasi linearizable and hence not linearizable either.

The pseudocode for checking quasi linearizability is shown in Algorithm 1, which takes a concurrent history ρ and a quasi factor K as input and returns either TRUE (quasi linearizable) or FALSE (not quasi linearizable). For ease of presentation, we assume that $O_o(d) = K$ for all subsets $d \in \mathit{Powerset}(\mathit{Domain}(o))$, where K is an integer constant. The main challenge in Algorithm 1 is to generate the set Φ of sequentializations of the given history ρ and the set Ψ of quasi permutations of each $\rho' \in \Phi$. The first step will be explained in the remainder of this section. The second step, which is significantly more involved, will be explained in the next section.

We now explain the detailed algorithm for computing the set Φ of sequentializations for the given history ρ. The computation is carried out by Subroutine `compute_sequentializations`(ρ). Let a history $\rho_0 = \varphi\ \mathtt{inv}_1\ \mathtt{inv}_2\ \phi\ \mathtt{resp}_1\ \psi\ \mathtt{resp}_2\ \ldots$ where φ, ϕ and ψ are arbitrary subsequences and $\mathtt{inv}_1, \mathtt{inv}_2$ are the invocation events of the first two overlapping method calls. We will replace ρ_0 in Φ with the new histories ρ_1 and ρ_2. In other words, for any two method call pairs $(\mathtt{inv}_i, \mathtt{resp}_i)$ and $(\mathtt{inv}_j, \mathtt{resp}_j)$ in ρ, if they do not overlap in time, meaning that either $\mathtt{resp}_i <_\rho \mathtt{inv}_j$ or $\mathtt{resp}_j <_\rho \mathtt{inv}_i$, we will keep this execution order. But if they overlap in time, we will generate two new histories, where one has $\mathtt{resp}_i <_\rho \mathtt{inv}_j$ and the other has $\mathtt{resp}_j <_\rho \mathtt{inv}_i$.

Example. Consider the history in Figure 2.6 (left). The first two overlapping calls start with $\mathtt{inv}_1$ and $\mathtt{inv}_2$, respectively.

- First, we construct a new history where $(\mathtt{inv}_1, \mathtt{resp}_1)$ is moved ahead of $(\mathtt{inv}_2, \mathtt{resp}_2)$. This is straightforward because, by the time we identify $\mathtt{inv}_1$ and $\mathtt{inv}_2$, we can continue to traverse the event sequence to find $\mathtt{resp}_1$ in ρ_0 and then move it ahead of event $\mathtt{inv}_2$.

Algorithm 1 Checking the quasi linearizability of the concurrent history ρ with respect to the quasi factor K.

```
 1: check_quasi_linearizability ( ρ, K )
 2: {
 3:     Φ ← compute_sequentializations( ρ );
 4:     for each ( ρ′ ∈ Φ ) {
 5:         if ( ρ′ ∈ spec(o) ) return TRUE;
 6:         Ψ ← compute_quasi_permutations( ρ′, K );
 7:         for each ( ρ″ ∈ Ψ ) {
 8:             if ( ρ″ ∈ spec(o) ) return TRUE;
 9:         }
10:     }
11:     return FALSE;
12: }
13: compute_sequentializations ( ρ )
14: {
15:     Φ ← {ρ};
16:     while ( ∃ a concurrent history ρ0 ∈ Φ ) {
17:         Let ρ0 = φ inv1 inv2 ϕ resp1 ψ resp2 ...;
18:         ρ1 ← φ inv1 resp1 inv2 ϕ ψ resp2 ...;
19:         ρ2 ← φ inv2 resp2 inv1 ϕ resp1 ψ ...;
20:         Φ ← Φ ∪ {ρ1, ρ2} \ {ρ0};
21:     }
22:     return Φ;
23: }
24: compute_quasi_permutations ( ρ′, K )
25: {
26:     Ψ ← { };
27:     state_stack ← first_run( ρ′, K );
28:     while ( TRUE ) {
29:         ρ″ ← backtrack_run ( state_stack, ρ′ );
30:         if ( ρ″ = = null ) break;
31:         Ψ ← Ψ ∪ {ρ″};
32:     }
33:     return Ψ;
34: }
```

Since the resulting History 1 still has overlapping method calls, we repeat the process in the next iteration.

- Second, we construct a new history by moving $(\texttt{inv}_2, \texttt{resp}_2)$ ahead of $(\texttt{inv}_1, \texttt{resp}_1)$. This is a little more involved because there can be many other method calls of Thread T_1 that are executed between $\texttt{inv}_2$ and $\texttt{resp}_2$. We take all these events between $\texttt{inv}_1$ and $\texttt{resp}_2$, and move them after $\texttt{resp}_2$. In this example, the new history is History 2.

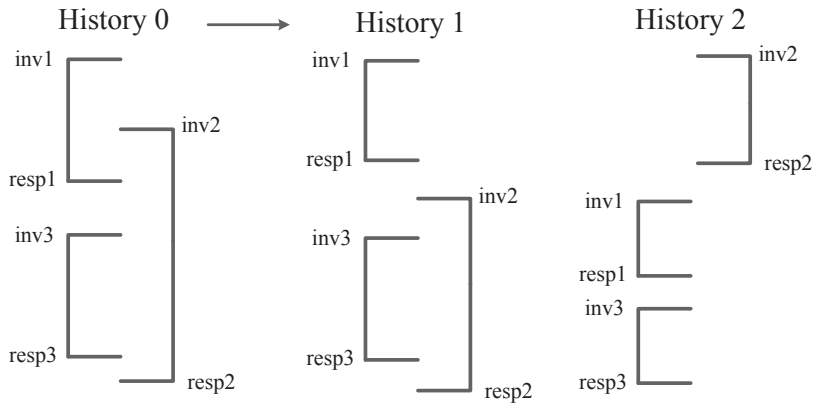


Figure 2.6: Example: Computing *sequentialization*s of a given concurrent history by repeatedly sequentializing the first two overlapping method calls denoted by `(inv1,resp1)` and `(inv2,resp2)`.

The complexity of `compute_sequentializations`(ρ) depends on the length of the input history ρ, as well as the number of overlapping method calls. Let M denote the length of the history ρ and L denote the number of overlapping method calls (where $L \leq M$). The complexity for computing all sequentializations of ρ is $O(M \times 2^L)$ in the worst case. In practice, however, this subroutine will not become a performance bottleneck for two reasons. First, to expose linearizability violations, small test programs with few concurrent method calls often suffice, which means that M is small. Second, the input to our method, $\rho = h|o$, is a subsequence of the original history projected to the object o, thereby consisting of only the events related to object o. Since linearizability is inherently compositional, we can check it for each individual object in isolation.

After computing the set Φ of sequentializations, we check if any $\rho' \in \Phi$ is a legal sequential history, as shown at Line 4 in Algorithm 1. According to Definition 1, as long as one sequentialization $\rho' \in \Phi$ is a legal sequential history, ρ is linearizable, which means that it is also quasi linearizable. Otherwise, ρ is not linearizable (but may still be quasi linearizable).

2.4 Checking for Quasi Linearizability

To check whether a sequentialization $\rho' \in \Phi$ is quasi linearizable, we need to invoke Subroutine `compute_quasi_permutations`(ρ', K). As shown in Algorithm 1, the subroutine consists of two steps. In the first step, `first_run` is invoked to construct a doubly linked list to hold the sequence of states connected by events in ρ', denoted `state_stack`: $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots s_n \xrightarrow{e_n}$. Each state s_i, where $i = 1, \ldots, n$, represents an abstract state of the object o. Subroutine `first_run` also fills up the fields of each state with the information needed later to generate the quasi permutations. In the second step, we generate quasi permutations of $\rho' \in \Psi$, one at a time, by calling `backtrack_run`.

2.4.1 Example: Constructing Quasi Permutations

We generate the quasi permutations by reshuffling the events in ρ' to form new histories. More specifically, we compute all possible permutations of ρ', denoted $\{\rho''\}$, such that the distance between ρ' and ρ'' is bounded by the quasi factor K. Our method for constructing the quasi permutations follows the *strict out-of-order* semantics as defined in [4, 48]. Consider queues as the example. A *strict out-of-order* k-quasi permutation consists of two restrictions:

- **Restriction 1**: each `deq` is allowed to return a value that is at most k steps away from the head node.
- **Restriction 2**: the first data element (in head node) must be returned by one of the first k `deq` operations.

To illustrate the *strict out-of-order* definition, consider the 1-quasi queue below, where the sequentialized history ρ' is `deq()=1,deq()=2,deq()=3`.

```
History 0:    deq(1) --> deq(2) --> deq(3)

                                                  Res1   Res2
```

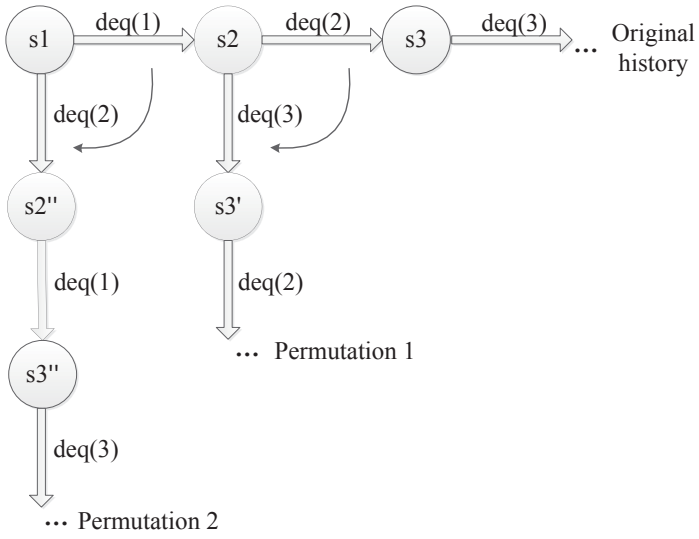


Figure 2.7: An example search tree for generating all 1-quasi permutations of the input sequence `deq(1);deq(2);deq(3)`.

```
History 1:    deq(2) --> deq(1) --> deq(3)      ok     ok
History 2:    deq(1) --> deq(3) --> deq(2)      ok     ok
History 3:    deq(3) --> deq(1) --> deq(2)      NO     ok
History 4:    deq(2) --> deq(3) --> deq(1)      ok     NO
History 5:    deq(3) --> deq(2) --> deq(1)      NO     NO
```

The input history can be arbitrarily re-shuffled to produce five additional histories, of which only History 1 and History 2 satisfy the above two restrictions of the *strict out-of-order* 1-quasi permutation. They are the desired quasi permutations of ρ' whereas the others are not. In particular, History 3 violates Restriction 1 because the first `deq` returns the value that is two steps away from the head. History 4 violates Restriction 2 because the head value is returned by the third `deq` operation, which is too late. History 5 violates both restrictions.

We compute the quasi permutations using a depth-first search (DFS) of the abstract states. For the above example, this process is illustrated in Figure 2.7, where the initial run is assumed to be $s_1 \xrightarrow{\texttt{deq(1)}} s_2 \xrightarrow{\texttt{deq(2)}} s_3 \xrightarrow{\texttt{deq(3)}}$.

- In the initial run, we construct the state stack that holds the initial history. Then we find the last backtrack state, which is state s_2, where we can execute `deq(3)` instead of `deq(2)`. This leads to the second run $s_1 \xrightarrow{\texttt{deq(1)}} s_2 \xrightarrow{\texttt{deq(3)}} s'_3 \xrightarrow{\texttt{deq(2)}}$.
- In the second run, we again find the last backtrack state, which is s_1, and execute `deq(2)` instead of `deq(1)`. This leads to the third run $s_1 \xrightarrow{\texttt{deq(2)}} s''_2 \xrightarrow{\texttt{deq(1)}} s''_3 \xrightarrow{\texttt{deq(3)}}$.
- In the third run, we can no longer find any backtrack state. Therefore, the procedure terminates. We cannot generate a new run by choosing `deq(3)` in state s_1, because it would violate Restriction 1. We cannot generate a new run by choosing `deq(3)` in state s''_2 either, because it would violate Restriction 2.

An important feature of our algorithm is that it directly constructs valid permutations while skipping the invalid ones, as opposed to constructing all permutations and then filtering out the invalid ones. In the running example, for instance, our method will directly generate History 1 and History 2 from History 0, while skipping 3, 4, and 5.

2.4.2 Elementary Data Structures

To enforce the restrictions imposed by the *strict out-of-order* semantics, we need to add some fields into each state. In particular, we add an *enabled* field into each state to help enforce Restriction 1, and we add a *lateness* attribute into each enabled event to enforce Restriction 2.

State_stack: We store the sequence of states of the current run in a doubly linked list called state_stack. Executing a method call event moves the object from one state to another state. Each state s has the following fields:

- s.enabled is the set of events that can be executed at s;
- s.select is the event executed by the current history;
- s.done is the set of events executed at s by some previously explored permutations in the backtrack search;

- s.newly_enabled is the set of events that become enabled for the first time along the given history ρ'. The field is initialized by the first run, and is used to compute the s.enabled field in the subsequent runs.

Example 1: s`.newly_enabled`. The initial state has at most $(K+1)$ events in its newly_enabled field, where K is the quasi factor. Every other state has at most one event in this newly_enabled field. For the given history `deq(1);deq(2);deq(3)` and quasi factor 1, we have

s_1.newly_enabled={deq(1),deq(2)}	$\geq$1 events in the initial state
s_2.newly_enabled={deq(3)}	at most one event
s_3.newly_enabled={ }	at most one event

In other words, each event will appear in the newly_enabled field of the state that is precisely K steps ahead of its original state in ρ'. We will enforce Restriction 1 with the help of the newly_enabled field.

Example 2: s`.enabled` and s`.done`. For the above example,

s_1.enabled={deq(1),deq(2)}	s_1.done={deq(1)}
s_2.enabled={deq(2),deq(3)}	s_2.done={deq(2)}
s_3.enabled={deq(3)}	s_3.done={deq(3)}

Both `deq(1)` and `deq(2)` are in s_1.enabled, but only `deq(1)` is in s_1.done because it is executed in the current run. Since the set (s.enabled\s.done) is not empty for both s_1 and s_2, we have two backtrack states. After backtracking to s_2 and executing `deq(3)`, we create a new permutation `deq(1);deq(3);deq(2)`. Similarly, after backtracking to s_1 and executing `deq(2)`, we create a new permutation `deq(2);deq(1);deq(3)`.

For permutation `deq(2);deq(1);deq(3)`, the enabled and done fields will be changed to the following:

s_1.enabled={deq(1),deq(2)}	s_1.done={deq(1),deq(2)}
s_2''.enabled={deq(1),deq(3)}	s_2''.done={deq(1)}
s_3''.enabled={deq(3)}	s_3''.done={deq3}

Although the set (s_2''.enabled\s_2''.done) is not empty, we cannot create the new permutation `deq(2);deq(3);deq(1)` because `deq(1)` would be out-of-order by two steps. We avoid generating such permutations by leveraging the lateness attribute that is added into every enabled event.

Lateness attribute: Each event in s.enabled has a lateness attribute, indicating how many steps this event is later than its original occurrence in ρ'. It represents how many steps this event can be postponed further in the current permutation.

```
s[i-k]              lateness(e) = -k
...
s[i].select = e     lateness(e) =  0
...
s[i+k]              lateness(e) =  k
```

Example 3: Consider the example above, where event e is executed in state s_i of the given history. For k-quasi permutations, the earliest state where e may be executed is s_{i-k}, and the latest state where e may be executed is s_{i+k}. The lateness attribute of event e in state s_{i-k} is $-k$, meaning that it may be postponed for at most $k-(-k)=2k$ steps. The lateness of e in state s_{i+k} is k, meaning that e has reached the maximum lateness and therefore must be executed in this state.

Must-select event: This brings us to the important notion of must-select event. In s.enabled, if there does not exist any event whose lateness reaches k, all the enabled events can be postponed for at least one more step. In this case, we can randomly choose an event from the set (s.enabled \s.*done*) to execute. If there exists an event in s.enabled whose lateness is k, then we must execute this event in state s.

Example 4: If we backtrack from the current history `deq(1),deq(2),deq(3)` to state s_1 and then execute `deq(2)`, event `deq(1)` will have a lateness of 1 in state s_2'', meaning that it has reached the maximum delay allowed. Therefore, it has to be executed in state s_2.

s_1.lateness={deq(1):lateness=0, deq(2):lateness=-1}
s_2''.lateness={deq(1):lateness=1, deq(3):lateness=-1}
s_3''.lateness={deq(3):lateness=0}

The initial lateness is assigned to each enabled event when the event is added to s.enabled by `first_run`. Every time an event is not selected for execution in the current state, it will be inherited by the enabled field of the subsequent state. The lateness of this event is then increased by 1.

An important observation is that, in each state, there can be at most one *must-select* event. This is because the first run ρ' is a total order of events, which gives each event a different *lateness* value—by definition, their *expiration times* are all different.

2.4.3 Algorithm: Constructing K-Quasi Permutations

The pseudo code for generating quasi permutations of the given history ρ' is shown in Algorithm 2. Initializing the lateness attributes of the enabled events is performed by Subroutine `init_enabled_and_lateness`, which is called by `first_run`. The lateness attributes are then updated by `update_enabled_and_lateness`.

Each call to `backtrack_run` will return a new quasi permutation of ρ'. Inside this subroutine, we search for the last backtrack state s in state_stack. If such backtrack state s exists, we prepare the generation of a new permutation by resetting the fields of all subsequent states of s, while keeping their newly_enabled fields intact. Then we choose a previously unexplored event in s.enabled to execute.

The previously unexplored event in s.enabled is chosen by calling `pick_an_enabled_event`.

Algorithm 2 Generating K-quasi permutations for history ρ'.

```
1: first_run ( ρ', K ) {
2:     state_stack ← empty list;
3:     for each ( event ev in the sequence ρ' ) {
4:         s ← new state;
5:         state_stack.append( s );
6:         s.select ← ev;
7:         s.done ← {ev};
8:         init_enabled_and_lateness( s, ev, K );
9:     }
10:    return state_stack;
11: }
12: init_enabled_and_lateness ( s, ev, K ) {
13:    lateness ← 0;
14:    while( 1 ) {
15:        s.enabled.add( ⟨ev,lateness⟩ );
16:        if( lateness == −k || s.prev == null ) {
17:            s.newly_enabled.add( ⟨ev, lateness⟩ );
18:            break;
19:        }
20:        lateness−−;
21:        s ← s.prev in state_stack;
22:    }
23: }
24: backtrack_run ( state_stack, ρ ) {
25:    Let s be the last state in state_stack such that
26:        pick_an_enabled_event( s ) ≠ null;
27:    if( such s does not exist )
28:        return null;
29:    for each( state after s in state_stack ) {
30:        reset s.select, s.done, and s.enabled,
31:        but keep s.newly_enabled;
32:    }
33:    while( s ≠ null ) {
34:        ev ←pick_an_enabled_event ( s );
35:        s.select ← ev;
36:        s.done ← {ev};
37:        s ← s.next;
38:        update_enabled_and_lateness( s );
39:    }
40:    return (sequence of selected events in state_stack);
41: }
42: pick_an_enabled_event ( s ) {
43:    if( ∃⟨ev, lateness⟩ ∈ s.enabled && lateness = k) {
44:        if( ev ∉ s.done )     // must-select event
45:            return ev;
46:        else
47:            return null;
48:    }
49:    if( ∃⟨ev, lateness⟩ ∈ s.enabled && ev ∉ s.done) )
50:        return ev;
51:    else
52:        return null;
53: }
54: update_enabled_and_lateness ( s ) {
55:    p ← s.prev;
56:    if( s or p do not exist )
57:        return;
58:    s.enabled ← { };
59:    for each( ⟨ev, lateness⟩ ∈ p.enabled && ev ∉ p.done) {
60:        s.enabled.add( ⟨ev, lateness − −⟩ );
61:    }
62:    for each( ⟨ev, lateness⟩ ∈ s.newly_enabled ) {
63:        s.enabled.add( ⟨ev, lateness⟩ );
64:    }
65: }
```

If there exists a must-select event in s.enabled whose lateness reaches k, then it must be chosen. Otherwise, we choose an event from the set (s.enabled $\backslash s$.done) arbitrarily. We use `update_enabled_and_lateness` to fill up the events in s.enabled. For events that are inherited from the previous state's enabled set, we increase their lateness by one. We iterate until the last state is reached. At this time, we have computed a new quasi permutation of ρ'.

The complexity of `compute_quasi_permutations()` depends on the length of the input history ρ', denoted M, as well as the quasi factor K. The overall complexity is $O((1+K)!^{M/(1+K)})$, where $(1+K)!$ is the number of permutations of $(1+K)$ events, and $M/(1+K)$ is the number of groups of size $(1+K)$ in a history of M events. In practice, this subroutine will not become a performance bottleneck because test programs with small M often suffice in revealing linearizability violations. In addition, the input $\rho' = h|o$ is a project of the original history h to each individual object o, which further reduces the size of the input.

2.4.4 Discussions

The real performance bottleneck in practice is due to the potentially large number of concurrent histories that need to be fed to our method as input, as opposed to the computational overhead of Phases 1 and 2 within our method. The reason why we may have a large number of concurrent histories is because the number is exponential in the number of *low-level concurrent operations* in the test program. Typically, the number of these low-level operations, denoted N, can be significantly larger than M, the number of method calls in the same history, because each method has to be implemented by many low-level operations such as Load/Store over shared memory and thread synchronizations.

In our experiments, we try to avoid the performance bottleneck by reducing the number of concurrent histories, primarily through the use of small test programs as input, and advanced exploration heuristics (such as context bounding) in the Inspect tool.

Our method is geared toward bug hunting. Whenever we find a concurrent history ρ that is not quasi linearizable, it is guaranteed to be a real violation. Furthermore, at this moment, the erroneous execution trace that gives rise to ρ has been logged into a disk file by the underlying Inspect tool. The execution trace contains all the method invocation and response events in ρ, as well as the low-level concurrent operations, such as Load/Store operations and thread synchronizations. Such trace can be provided to the user as the debugging aid. As an example, we will show in Section 2.5.4 how it can help the user diagnose a real bug found in the *Scal* benchmarks.

However, since our method implements the L1 and L2 checks but not the L3 check as defined in Section 2.2, even if all concurrent histories of the test program are quasi linearizable, we cannot conclude that the concurrent data structure itself is quasi linearizable.

Furthermore, when checking for quasi linearizability, our runtime checking framework has the capability of generating test programs (harness) that are *well-formed*; that is, the number of `enq` operations is equal to the number of `deq` operations. If the test program is provided by the user, then it is the user's responsibility to ensure this well-formedness. This is important because, if the test program is not well-formed, there may be *out-of-thin-air* events. Below is an example.

```
Thread 1  Thread 2       Hist1       Hist2       Hist3
--------  --------
enq(3)    enq(5)         enq(3)      enq(5)      enq(3)
enq(4)    deq()          ...         ...         enq(4)
...       ...            ...         ...         enq(5)
--------  --------       deq()=3     deq()=5     deq()=4
```

Here, the sequential specification is {`Hist1,Hist2`}. In both legal sequential histories, either `deq()=3` or `deq()=5`. However, the `deq` value can never be 4. This is unfortunate, because `Hist3` is 1-quasi linearizable but cannot match any of the two legal sequential histories (`Hist1` or `Hist2`) because it has `deq()=4`. This problem can be avoided by requiring the test program given to our method to be well-formed. For example, by adding two more `deq` calls

to the end of the main thread, we can avoid the aforementioned *out-of-thin-air* events.

2.5 Experiments

We have implemented our new quasi linearizability checking method in a software tool based on the LLVM platform for code instrumentation and *Inspect* for systematically generating interleaved executions. Our tool, called *Round-Up*, can handle C/C++ code of concurrent data structures on the Linux/PThreads platform. We have extended the original implementation of *Inspect* [153, 154] by adding the support for GNU built-in atomic functions for direct access of shared memory, since in practice, they are frequently used in the low-level code for implementing concurrent data structures.

We have conducted experiments on a set of concurrent data structures [4, 69, 99, 47, 69, 48] including both standard and quasi linearizable queues, stacks, and priority queues. For some of these concurrent data structures, there are several variants, each of which uses a different implementation scheme. These benchmark examples fall into two sets.

2.5.1 Results on the First Set of Benchmarks

The characteristics of the first set of benchmark programs are shown in Table 2.1. The first three columns list the name of the data structure, a short description, and the number of lines of code. The next two columns show whether it is linearizable and quasi linearizable. The last column provides a list of the relevant methods.

Table 2.2 shows the results for checking standard linearizability. The first four columns show the statistics of the test program, including the name, the number of threads (concurrent/total), the number of method calls, and whether linearizability violations exist. For example, 3*4+1 in the *calls* column means that one child thread issued 3 method calls, the other child thread issued 4 method calls, and the main thread issued 1 method call. The next two

Table 2.1: The statistics of the benchmark examples.

Class	**Description**	**LOC**	**Linearizable**	**Quasi-Lin**	**Methods checked**
IQueue	buggy queue, deq may remove null even if not empty	154	No	NO	enq(int), deq()
Herlihy/Wing queue	correct normal queue	109	YES	YES	enq(int), deq()
Quasi Queue	correct quasi queue	464	NO	YES	enq(int), deq()
Quasi Queue b1	deq removes value more than k away from head	704	NO	NO	enq(int), deq()
Quasi Queue b2	deq removes values that have been removed before	401	NO	NO	enq(int), deq()
Quasi Queue b3	deq null even the queue is not empty	427	NO	NO	enq(int), deq()
Quasi Stack b1	pop null even if the stack is not empty	487	NO	NO	push(int), pop()
Quasi Stack b2	pop removes values move than k away from the tail	403	NO	NO	push(int), pop()
Quasi Stack	linearizable, and hence quasi linearizable	403	YES	YES	push(int), pop()
Quasi Priority Queue	implementation of quasi priority queue	508	NO	YES	enq(int, int), deqMin()
Quasi Priority Queue b2	deqMin removes value more than k away from head	537	NO	NO	enq(int, int), deqMin()

Table 2.2: Results of checking standard linearizability on concurrent data structures.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	sequentialization	time (seconds)
IQueue	2/3	2*2+0	YES	3	0.1	2/6	13	0.3
Herlihy/Wing queue	2/3	2*2+0	NO	3	0.1	0/4	9	0.2
Quasi Queue	2/3	2*2+4	YES	6	0.2	16/16	61	2.5
Quasi Queue	2/3	3*3+4	YES	20	1.1	64/64	505	43.7
Quasi Queue	2/3	2*3+3	YES	10	0.4	24/32	169	8.3
Quasi Queue	2/3	3*3+2	YES	20	0.8	108/118	1033	1m23s
Quasi Queue	2/3	3*4+1	YES	35	1.6	149/198	2260	5m8s
Quasi Queue	2/3	4*4+0	YES	70	2.6	274/476	8484	37m34s
Quasi Queue b1	2/3	2*2+4	YES	6	0.3	91/91	409	17.8
Quasi Queue b2	2/3	2*2+4	YES	6	0.3	91/91	409	18.1
Quasi Queue b3	2/3	2*2+4	YES	6	0.3	141/141	653	26.9
Quasi Stack b1	2/3	2*2+4	YES	6	0.3	9/9	34	1.6
Quasi Stack b2	2/3	2*2+4	YES	6	0.3	16/16	61	2.5
Quasi Stack	2/3	2*2+4	NO	6	0.3	0/16	61	2.5
Quasi Priority Queue	2/3	2*2+4	YES	6	0.5	16/16	61	4.9
Quasi Priority Queue b2	2/3	2*2+4	YES	6	0.5	125/125	532	27.0

columns show the statistics of Phase 1, consisting of the number of sequential histories and the time for generating these sequential histories. The last three columns show the statistics of Phase 2, consisting of the number of concurrent histories (buggy/total), the total number of sequentializations, and the time for checking them. In all test cases, our method was able to correctly detect the linearizability violations.

Table 2.3 shows the experimental results for checking quasi linearizability. The first four columns show the statistics of the test program. The next two columns show the statistics of Phase 1, and the last three columns show the statistics of Phase 2, consisting of the number of concurrent histories (buggy/total), the total number of quasi permutations, and the time for generating and checking them. In all test cases, we set the quasi factor to 2 unless specified otherwise in the test program name (e.g., qfactor=3).

Table 2.3: Results of checking quasi linearizability on concurrent data structures.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	permutation	time (seconds)
Quasi Queue	2/3	2*2+4	NO	6	0.2	0/16	1708	2.9
Quasi Queue	2/3	3*3+4	NO	20	1.1	0/64	73730	5m33s
Quasi Queue	2/3	2*3+3	NO	10	0.4	0/32	4732	9.6
Quasi Queue	2/3	3*3+2	NO	20	0.8	0/118	28924	1m34s
Quasi Queue	2/3	3*4+1	NO	35	1.6	0/198	63280	5m40s
Quasi Queue	2/3	4*4+0	NO	70	2.6	0/476	237552	40m56s
Quasi Queue (qfactor=3)	2/3	2*3+3	NO	10	0.4	0/32	8112	14.9
Quasi Queue (qfactor=3)	2/3	3*3+2	NO	20	0.8	0/118	49584	2m36s
Quasi Queue (qfactor=3)	2/3	3*4+1	NO	35	1.6	0/198	108480	10m15s
Quasi Queue (qfactor=3)	2/3	4*4+0	NO	70	2.6	0/476	407232	69m32s
Quasi Queue b1	2/3	2*2+4	YES	6	0.3	41/91	11452	20.1
Quasi Queue b2	2/3	2*2+4	YES	6	0.3	91/91	11452	20.2
Quasi Queue b3	2/3	2*2+4	YES	6	0.3	73/141	18284	31.0
Quasi Stack b1	2/3	2*2+4	YES	6	0.3	9/9	2108	3.5
Quasi Stack b2	2/3	2*2+4	YES	6	0.3	6/16	1708	2.8
Quasi Stack b3	2/3	2*2+4	NO	6	0.3	0/16	1708	2.8
Quasi Priority Queue	2/3	2*2+4	NO	6	0.5	0/16	1708	4.7
Quasi Priority Queue b2	2/3	2*2+4	YES	6	0.5	54/125	6384	20.0

Our method was able to detect all real (quasi) linearizability violations in fairly small test programs. This is consistent with the experience of Burckhart *et al.* [17] in evaluating their Line-Up tool for checking standard (but not quasi) linearizability. This is due to the particular application of checking the implementation of concurrent data structures. Although the number of method calls in the test program is small, the underlying low-level shared memory operations can still be many. This leads to a rich set of very subtle interactions between the low-level memory accessing instructions. In such cases, the buggy execution can be uncovered by checking a test program with only a relatively small number of threads, method calls, and context switches.

2.5.2 Results on the *Scal* Benchmarks

We have also conducted experiments on a set of recently released high-performance concurrent objects in the *Scal* suite [112]. The characteristics of this second set of benchmark programs are shown in Table 2.4. The format of this table is the same as Table 2.1. For a more detailed description of the individual algorithms, please refer to the original papers where these concurrent data structures are published [48, 69, 68].

Table 2.4: The statistics of the Scal [112] benchmark examples (total LoC of *Scal* is 5,973).

Class	Description	LOC	Linearizable	Quasi-Lin	Methods checked
sl-queue	singly-linked list based single-threaded queue	73	NO	NO	enq, deq
t-stack	concurrent stack by R. K. Treiber	109	YES	YES	push, pop
ms-queue	concurrent queue by M. Michael and M. Scott	250	YES	YES	enq, deq
rd-queue	random dequeued queue by Y. Afek, G. Korland, and E. Yanovsky	162	NO	YES	enq, deq
bk-queue	bounded k-FIFO queue by Y. Afek, G. Korland, and E. Yanovsky	263	NO	YES	enq, deq
ubk-queue	unbounded k-FIFO queue by C.M. Kirsch, M. Lippautz, and H. Payer	259	NO	YES	enq, deq
k-stack	k-stack by T. A. Henzinger, C. M. Kirsch, H. Payer, and A. Sokolova	337	NO	NO	push, pop

Table 2.5 shows our experimental results for applying *Round-Up* to this set of benchmarks. In addition to obtaining the performance data, we have successfully detected two real linearizability violations in the *Scal* suite, one of which is a known violation whereas the other is a previously unknown programming error. In particular, *sl-queue* is a queue designed for high-performance applications, but it is not thread safe and therefore is not linearizable. Note that the fact that *sl-queue* is not linearizable is known.

In contrast, *k-stack* is designed to be quasi linearizable, but due to an ABA bug, the data structure implementation is not quasi linearizable. In fact, it is not even linearizable since the behavior may violate the standard functional specification of a stack. (An ABA problem [49] occurs in a multithreaded program when a memory location is read twice, has the same values for both reads, and therefore appears to indicate "nothing has changed." However, another thread may have interleaved in between the two reads, changed the value, did some work, and then changed the value back.)

Our tool is able to quickly detect the linearizability violation in *sl-queue* and the quasi linearizability violation in *k-stack*. We have reported the violation in *k-stack* to the *Scal* developers, who have confirmed that it is indeed a bug.

It is worth pointing out that existing concurrency bug finding tools such as data-race and atomicity violation detectors are not effective for checking low-level C/C++ code that implements most of the highly concurrent data structures. These bug detectors are designed primarily for checking application level code. Furthermore, they are often based on the lockset analysis and condition variable analysis. Although locks and condition variables are widely used in writing application level code, they are rarely used in implementing concur-

Table 2.5: Results of checking quasi linearizability for the Scal [112] benchmark examples.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	permutation	time (seconds)
sl-queue (enq+deq)	2/3	1*1+12	NO	2	0.38	0/2	1032	1.61
sl-queue (enq+enq)	2/3	1*1+12	YES	2	0.37	1/2	1032	1.78
sl-queue (deq+deq)	2/3	1*1+12	YES	2	0.37	4/8	5160	7.21
t-stack (push+pop)	2/3	1*1+12	NO	2	0.58	0/8	5160	7.77
t-stack (push+push)	2/3	1*1+12	NO	2	0.59	0/8	5160	7.78
t-stack (pop+pop)	2/3	1*1+12	NO	2	0.56	0/8	5160	7.63
ms-queue (enq+deq)	2/3	1*1+12	NO	2	0.76	0/3	1720	2.94
ms-queue (enq+enq)	2/3	1*1+12	NO	2	0.79	0/31	20984	32.47
ms-queue (deq+deq)	2/3	1*1+12	NO	2	0.76	0/12	7912	12.33
rd-queue (enq+deq)	2/3	1*1+12	NO	2	0.90	0/5	3096	5.97
rd-queue (enq+enq)	2/3	1*1+12	NO	2	0.91	0/31	20984	37.83
rd-queue (deq+deq)	2/3	1*1+12	NO	2	0.87	0/4	2408	7.31
bk-queue (enq+deq)	2/3	1*1+12	NO	2	1.27	0/29	19608	34.45
bk-queue (enq+enq)	2/3	1*1+12	NO	2	1.30	0/41	27864	50.98
bk-queue (deq+deq)	2/3	1*1+12	NO	2	1.22	0/24	16168	28.11
ubk-queue (enq+deq)	2/3	1*1+12	NO	2	2.42	0/21	14104	35.71
ubk-queue (enq+enq)	2/3	1*1+12	NO	2	1.86	0/41	27864	61.94
ubk-queue (deq+deq)	2/3	1*1+12	NO	2	1.84	0/52	35432	68.21
k-stack (push+pop)	2/3	1*1+12	YES	2	1.55	11/69	47128	1m29s
k-stack (push+push)	2/3	1*1+12	NO	2	1.69	0/12	7192	15.13
k-stack (pop+pop)	2/3	1*1+12	NO	2	1.62	0/8	5160	9.46

rent data structures. Synchronization in concurrent data structures may be implemented using atomic memory accesses. To the best of our knowledge, no prior method can directly check quantitative properties in such low-level C/C++ code.

2.5.3 Optimizations to Reduce Runtime Overhead

To further improve the runtime performance, we have applied the following optimizations in Phases 1 and 2, to reduce the number of traces generated from the sequential specification $spec(o)$, the set Φ of sequentializations, and the set Ψ of quasi permutations.

Specifically, our optimizations are as follows: (1) in Phase 1, we remove from $spec(o)$ all identical serial histories generated by the Inspect tool by simply comparing their textual representations; (2) in Phase 2, we remove from the input all identical concurrent executions fed to our method—we say that two concurrent executions are identical if they give rise to the same history, even though they may differ in the low-level concurrent operations within the method calls; and (3) for each sequentialization $\rho' \in \Phi(\rho)$, or each quasi permutation

$\rho'' \in \Psi(\rho')$, we check if ρ' or ρ'' is identical to some sequentialization or quasi permutation of a previously explored concurrent history, and if the answer is yes, we skip it.

It is worth pointing out that, in our application, the length of the sequential and concurrent execution histories are often short. Therefore, a straightforward implementation of the above algorithm for identifying redundant histories already works well in practice. For example, in Table 2.5, each execution history has at most 14 method calls (1+1+12), among which only two method calls are executed concurrently. In such case, the pair-wise comparison of execution histories as described above is fast. Specifically, compared to the cost of the other parts of our method, e.g., loading the executable to the main memory and performing the interleaved executions, the time spent on comparing these histories in memory is always negligible. This is the reason why, in Table 2.5, we only report the number of permutations and the total time.

Our evaluation of these optimizations on the *Scal* benchmarks shows that they can lead to a significant reduction in the number of permutations and the execution time. Our experimental results are shown in the scatter plots in Figure 2.8 and 2.9, which compare the number of permutations and the total time of our method with and without the optimizations. Note that the concrete time and the number of permutations, with optimizations, have already been reported in the last two columns of Table 2.5. The x-axis shows the number of permutations and the execution time without these optimizations, whereas the y-axis shows the number of permutations and the execution time with these optimizations. Here, points below the red diagonal lines are the winning cases for the optimizations. Note that both the x-axis and the y-axis are in logarithmic scale.

2.5.4 Generating Diagnosis Information

Our *Round-Up* tool guarantees that all the reported (quasi) linearizability violations are real violations; that is, they can appear in the actual program execution. To help the developer diagnose the reported violation, our tool leverages Inspect, the underlying concurrency test-

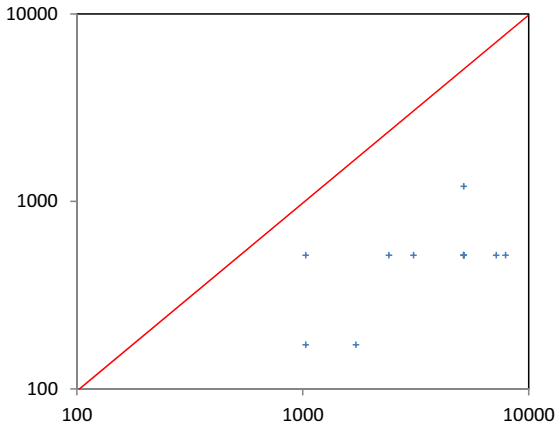


Figure 2.8: Results: Evaluating the impact of optimizations on the performance for *Scal* benchmarks [112]. We compare the *number of quasi permutations* generated by our method with optimizations (y-axis) and without optimizations (x-axis). Each + represents a test case. Test cases below the diagonal line are the winning cases for our method with optimizations.

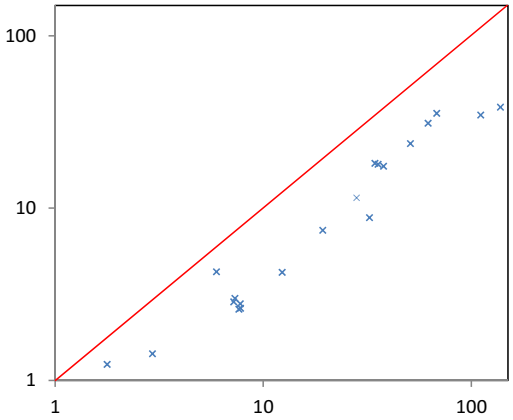


Figure 2.9: Results: Evaluating the impact of optimizations on the performance for *Scal* benchmarks [112]. We compare the *execution time (in seconds)* of our method with optimizations (y-axis) and without optimizations (x-axis). Each + represents a test case. Test cases below the diagonal line are the winning cases for our method with optimizations.

ing tool, to generate a detailed execution trace for each concurrent history. When a current history ρ is proved to be erroneous, i.e., neither linearizable nor quasi linearizable, the detailed execution trace will be provided to the user as debugging aid. In this execution trace, we have recorded not only the method invocation and response events in ρ but also the low-level operations by each thread, including Load/Store instructions over shared memory and

thread synchronizations. For more information on how such execution trace is generated, please refer to the Inspect tool [153, 154].

In terms of the ABA bug that we have detected in *k-stack*, for example, our tool shows that the violation occurs when one thread executes the `push` operation while another thread executes the `pop` operation concurrently. Due to erroneous thread interleaving, it is possible for the same data item to be added to the stack twice, despite the fact that the `push` operation is executed only once. In the remainder of this subsection, we shall explain this bug in more details.

First, we show in Algorithm 3 the simplified pseudo code of the `push` and `pop` methods of *k-stack* as implemented in the *Scal* suite. The `push()` method starts by putting the data item into the current segment and then trying to commit, by checking if the flag `segment→delete` is 0. The `pop()` method, which may be executed by another thread concurrently with `push()`, starts by setting the flag `segment→delete` to 1 and then trying to remove the segment if it is empty.

The bug happens when the top segment is empty, thread T_1 is trying to push a data item onto the stack, and at the same time, thread T_2 is trying to pop a data item from the stack. The erroneous execution trace is summarized as follows:

```
Thread 1                        Thread 2
------------------------        ------------------------
                                Pop:L10-12
Push:L3-L5
                                try_remove_segmt:L23
Committed:L17
                                try_remove_segmt:L24-L25
Committed:L18-L19
Push:L3-L6
------------------------        ------------------------
```

Algorithm 3 The simplified pseudo code of *k-stack* in *Scal*

```
 1: push(T item)
 2: {
 3:     while ( TRUE ) {
 4:         put item into the segment;
 5:         if ( committed() ) return TRUE;
 6:     }
 7: }
 8: pop()
 9: {
10:     while ( TRUE ) {
11:         try to find an item in the segment;
12:         if (not found) try_remove_segment();
13:     }
14: }
15: committed()
16: {
17:     if ( segment→delete==0 ) return TRUE;
18:     else if ( segment→delete==1 ) {...}
19:     return FALSE;
20: }
21: try_remove_segment()
22: {
23:     segment→delete=1;
24:     if ( is_empty(segment) ) {...}
25:     else  segment→delete=0;
26: }
```

From the above debugging information reported by our *Round-Up* tool, we can explain the erroneous thread interleaving as follows:

- **Step 1**: Since the stack is empty, inside method `pop`, thread T_2 cannot find any data item in the segment, and therefore invokes function `try_remove_segment()` to try to remove the segment;

- **Step 2**: Inside method `push`, thread T_1 pushes a data item into the segment, and then calls function `committed()` to make sure the data item is successfully added by waiting `committed()` to return true;

- **Step 3**: Inside function `try_remove_segment()`, thread T_2 marks the top segment as *deleted*;

- **Step 4**: Inside function `committed()`, thread T_1 fails the condition in Line 17;
- **Step 5**: Inside function `try_remove_segment()`, thread T_2 checks the segment again, and notices that the segment is not empty. Therefore, thread T_2 sets the top segment back to *not deleted*;
- **Step 6**: Inside function `committed()`, thread T_1 fails the condition in Line 18 and returns false. Therefore, method `push` returns to the beginning of the while loop, and pushes the same data item into the segment again.

As we can see, this bug requires complex interactions between the two threads to manifest and therefore is difficult to detect manually. However, it can be easily detected by our *Round-Up* tool, since our tool has the capability of systematically enumerating possible thread interleavings and then checking each of them for (quasi) linearizability violations.

2.6 Related Work

Quasi linearizability was first introduced by Afek *et al.* [4] as a way to trade off correctness for better runtime performance while implementing concurrent queues for highly parallel applications. Since then, the idea of quantitatively relaxing the consistency requirement of linearizability to improve runtime performance have been used in the design of various concurrent data structures [47, 69, 68, 99]. More recently, Henzinger *et al.* [48] generalized the idea by proposing a systematic and formal framework for obtaining new concurrent data structures by quantitatively relaxing existing ones.

Round-Up is the first runtime verification method for detecting quasi linearizability violations in the source code implementation of concurrent data structures. This chapter is an extended version of our recent work [160, 161] where we proposed the method for the first time. In this extended version, we have provided a more detailed description of the algorithms and presented more experimental results.

A closely related work is a model checking based method that we proposed [2, 1] for verifying quantitative relaxations of linearizability in *models* of concurrent systems. However, the method was designed for statically verifying finite-state models, not for dynamically checking the C/C++ code of concurrent data structure implementations. Although the Line-Up tool by Burckhardt *et al.* [17] also has the capability of dynamically checking the code of concurrent data structures, it can only check linearizability but not *quasi* linearizability. In contrast, the main contribution of this chapter is proposing a new method for checking quasi linearizability.

Besides Line-Up [17], there is a large body of work on statically verifying standard linearizability properties. For example, Liu et al. [79] verify standard linearizability by proving that an implementation model refines a specification model. Vechev et al. [129] use the SPIN model checker to verify linearizability in a Promela model. Cerný et al. [19] use automated abstractions together with model checking to verify linearizability properties in Java programs. There are also techniques for verifying linearizability by constructing mechanical proofs, often with manual intervention [127, 128, 164]. However, none of these methods can check quantitative relaxations of linearizability.

There are also static and runtime verification methods for other types of consistency conditions, including sequential consistency [76], quiescent consistency [12], and eventual consistency [131]. Some of these consistency conditions, in principle, may be used during software testing and verification to ensure the correctness of concurrent data structures. However, they are not as widely used as linearizability in this application domain. Furthermore, they do not have the same type of quantitative properties and the corresponding verification challenges as in *quasi* linearizability.

For checking application level code, which has significantly different characteristics from the low-level code that implements concurrent data structures, *serializability* and *atomicity* are the two frequently used correctness criteria. In the literature, there is a large body of work on detecting serializability and atomicity violations (e.g. [40, 36, 119] and [38, 152, 83, 139,

20, 37, 39, 134, 155, 135, 136, 120, 121, 119, 118, 111, 132, 63, 53, 64, 115, 54, 133]). It is worth pointing out that these bug finding methods differ from our method in that they are checking for different types of properties. In practice, atomicity and serializability have been used primarily at the shared memory read/write instruction level, whereas linearizability has been used primarily at the method API level. Furthermore, existing tools for detecting serializability and atomicity violations do not check for quantitative properties.

2.7 Discussions

We have presented a new algorithm for runtime verification of standard and quasi linearizability in concurrent data structures. Our method works directly on the C/C++ code and is fully automated, without requiring the user to write functional specifications or to annotate linearization points in the code. It also guarantees that all the reported violations are real violations. We have implemented the new algorithm in a software tool called *Round-Up*. Our experimental evaluation shows that *Round-Up* is effective in detecting quasi linearizability violations and in generating information for error diagnosis.

Chapter 3

Preventing Concurrency Related Type-State Violations

We propose a unified runtime mitigation framework for suppressing *concurrency* related type-state violations in multithreaded C/C++ applications. Bugs in multithreaded applications are notoriously difficult to detect and fix due to the interleaving explosion, i.e., the number of possible interleavings can be exponential in the number of concurrent operations. Due to scheduling nondeterminism, multiple runs of the same application may exhibit different behaviors. As a result, existing software tools are often inadequate in supporting automated diagnosis and prevention of concurrency bugs. Although there has been some work on mitigating concurrency bugs [107, 158, 85, 77, 145, 58, 78, 143], they focus primarily on the concurrent accesses at the load/store instruction level, thereby leading to a significant runtime overhead. Our new method, in contrast, focuses on method calls of the shared objects.

In mainstream object-oriented programming, an application can be divided into two parts: a library of objects and the client that utilizes these objects. While developing the client, the programmer needs to follow a set of usage rules that define the application programming interface (API). Type-state automaton can be used to specify the temporal aspects of the

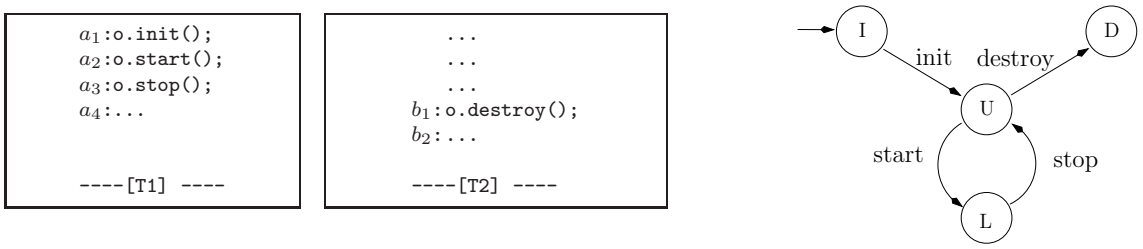


Figure 3.1: A client program and the type-state automaton. For ease of drawing, we have omitted the non-accepting state. However, there is an implicit edge from every state to the non-accepting state. For example, from state I, executing any method other than `init()` leads to the non-accepting state.

API that standard type systems cannot express. As illustrated in Figure 3.1 (right), the automaton is a graph where nodes are the abstract states of the object and edges are the method calls. A sequence of method calls is valid if and only if it corresponds to a path in the automaton.

When an object is shared by threads as in Figure 3.1 (left), its method call sequence depends on the thread execution order. This may lead to concurrency bugs. For example, the automaton in Figure 3.1 says that `start()` and `stop()` can be invoked only after the call to `init()` and before the call to `destroy()`. When thread T_1 completes before thread T_2 starts, the resulting call sequence satisfies this specification. However, when the thread interleaving is $a_1, b_1, a_2, a_3, \ldots$, the program violates this specification. In this case, the client program may crash due to the use of a destroyed object. Type-state violations of this kind appear frequently in real-world multithreaded applications, often marked in bug databases as *race conditions* by developers. Such *Heisenbugs* are often difficult to detect and diagnose, due to the often astronomically many thread interleavings in the programs.

Our method can be viewed as a runtime mitigation technique for suppressing such violations. We insert a layer between the client and the library to control method calls issued by the client to avoid the erroneous usage. For example, we may delay certain method calls if such runtime action can help enforce the desired client program behavior as specified in the type-state automaton. Alternatively, our method can be understood as hardening the client

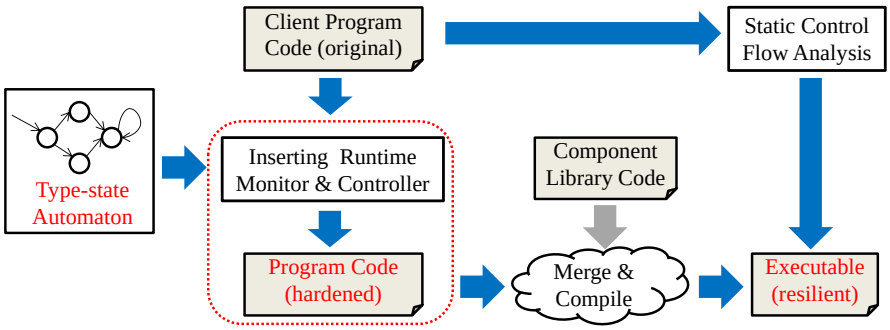


Figure 3.2: Preventing concurrency type-state errors.

program so that it emits only correct method call sequences.

The overall flow of our method is illustrated in Figure 3.2, which takes the client and the type-state automaton as input and generates a new client program and its control flow information as output. In the new client program, runtime monitoring and control routines such as `pre_call()` have been injected to add runtime analysis and failure prevention capabilities. The new client program is compiled with the original library to create an executable. At run time, the control strategy implemented inside `pre_call()` will be able to selectively delay certain method calls, to allow only correct method sequences as specified in the type-state automaton. The decision on *when to delay which thread* will depend on an online analysis of the type-state automaton, as well as its interaction with the control flow graph of the client program.

An important requirement for runtime error prevention is that the actions must *do no harm*. That is, they should not introduce new bugs. Traditionally, this requirement has been difficult to satisfy in *ad hoc* error mitigation methods. The main contribution of this chapter is to propose a unified theoretical framework within which various strategies can be analyzed to ensure that they are always safe. More specifically, we guarantee that our prevention strategies do not introduce new concurrency related bugs. Furthermore, when the static analysis is precise enough, our runtime mitigation method can always find the failure-free interleaving whenever such interleaving exists. The underlying theory for this framework is abstract model checking. To the best of our knowledge, this is so far the best result that can

be achieved by a runtime mitigation algorithm. To put it into context, most of the existing runtime mitigation methods cannot guarantee safety, i.e., the runtime mitigation actions may lead to artificial deadlocks, which have to be reported to the developers [86] or bailed out by adding a timeout.

Our method also differs from most of the existing methods for tolerating concurrency bugs [107, 158, 85, 77, 145, 58, 78, 143], in that they focus exclusively on low-level races. Since they often have to monitor the individual load/store instructions for shared memory accesses, the runtime performance overhead can be substantial. For example, the purely software based solution in AVIO-S [83] imposes an average slowdown of 25X (their AVIO-H imposes only 0.4-0.5% overhead but requires the support of a specially designed microprocessor hardware). In contrast, our purely software based method has to monitor only the relevant API calls involved in the shared objects. Therefore, it can maintain a significantly smaller performance overhead – our experimental evaluation shows an average slowdown of only 1.45% on C/C++ applications.

Concurrency related type-state violations are actually more common than they may appear despite the fact that they are not yet adequately addressed in the literature. On one hand, any low-level race on shared memory accesses eventually has to manifest itself at the method call level. On the other hand, it is almost impossible to have concurrency related type-state violations if there are no low-level races inside the methods. Therefore, type-state violations and low-level races should be considered as symptoms that are manifested by the same errors, but at different abstraction levels. For the purpose of runtime failure mitigation, we argue that raising the level of abstraction has the advantage of very low runtime overhead, as well as the availability of type-state automata as the clearly defined correctness conditions.

Having a clearly defined correctness condition is crucial to ensure that the mitigation actions do not introduce new bugs because it provides a clearly defined goal. In runtime mitigation of low-level races, such correctness conditions are generally not available. Instead, existing methods often resort to profiling and replay, leading to a notion of correctness that is at best

statistical, rather than categorical. For example, automatically inferred atomic regions may be bogus and data-races may be benign, which make it difficult to ensure the validity and safety of runtime prevention actions. Indeed, quite a few of these methods are not safe in that they may introduce artificial deadlocks. In contrast, our method relies on the type-state automaton, which formally specifies the correctness criterion, and therefore ensures that our mitigation actions are always safe. Finally, whereas replay based methods can only suppress previously encountered bugs, our method can suppress concurrency bugs that have not been exposed before.

We have implemented our new method in a software tool based on the LLVM framework for multithreaded C/C++ applications written using the POSIX Threads. We have evaluated it on a set of open-source applications with known bugs. Our experiments show that the new method is both efficient and effective in suppressing concurrency related type-state errors.

Our main contributions are summarized as follows:

- We propose a runtime method for suppressing concurrency related type-state violations in multithreaded applications. It offers a unified theoretical framework that ensures both the *safety* property and the *effectiveness* property of the runtime mitigation actions.
- We evaluate our method on a set of multithreaded C/C++ applications with known bugs. Our experimental results show that the new method is effective in suppressing these bugs and at the same time, imposes only an average performance overhead of 1.45%.

3.1 Motivating Examples

We provide an overview of the new runtime analysis and mitigation method by using the example in Figure 3.1. Recall that the buggy program involves at least the two threads in

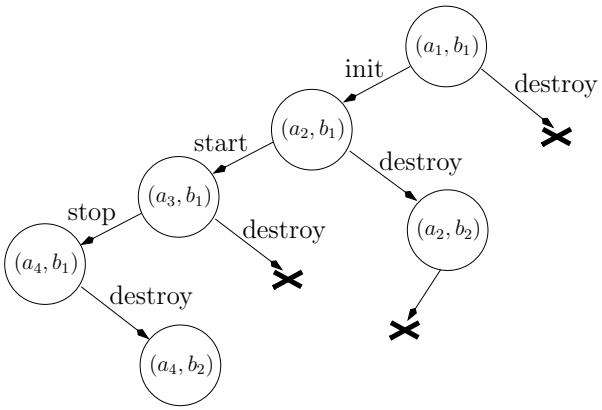


Figure 3.3: The combined state transition system for the client program M and the automaton A in Figure 3.1. Each cross represents a bad automaton state.

Figure 3.1, and possibly also many other threads that run concurrently with them. The intended interleaving of the two threads is $a_1, a_2, a_3, \ldots, b_1, b_2$, but the programmer fails to prevent the erroneous interleavings such as $b_1, b_2, a_1, \ldots$, which can cause the type-state violation.

To analyze all possible interleavings of the two threads, in principle, we can represent each thread as a state transition system. Let M_1 be the state transition system of thread T_1, and M_2 be the state transition system of thread T_2. The combined system $M = M_1 \times M_2$ is an interleaved composition of the two individual transition systems following the standard notion in the literature [51]. Figure 3.3 shows the combined system M, where nodes are the global program states and edges may be labeled by method calls from threads T_1 or T_2.

Each path in M represents an interleaving. For example, from state (a_1, b_1), we can execute T_1's `o.init()` to reach (a_2, b_1), or execute thread T_2's `o.destroy()` to reach (a_1, b_2). Similarly, from state (a_2, b_1), we can execute thread T_1's `o.start()` to reach (a_3, b_1), or execute thread T_2's `o.destroy()` to reach (a_2, b_2). There are four possible interleavings in Figure 3.3. According to the type-state automaton in Figure 3.1 (right), only the left-most interleaving is free of type-state violation.

Our method injects a control layer between the client and the library. As illustrated in

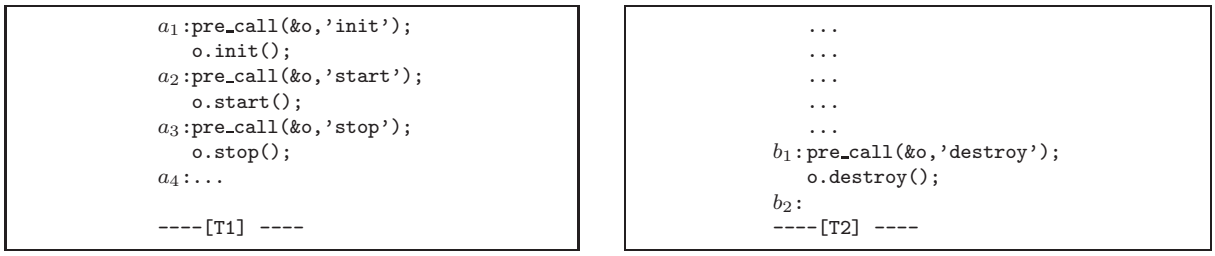


Figure 3.4: An instrumented client program with `pre_call()` controlling the execution of method calls.

Figure 3.4, we instrument the client program by inserting a `pre_call()` routine before every method call of the shared object in the client program. By passing the object address and method name as parameters at run time, we know which method is being called by which thread. Our mitigation strategy is implemented in `pre_call()`, to selectively delay certain method calls. In this example, when `o.destroy()` is about to execute, and the other thread has not yet finished executing the methods of the same object o, we will postpone the execution of `o.destroy()`.

Postponing the execution of the method call can be implemented by forcing thread T_2 to spin until the global state becomes (a_4, b_1). We invoke `usleep` in a spinning while-loop in `pre_call()`, which releases the CPU to execute other threads.

To minimize the performance overhead, the runtime mitigation decisions need to be made by a distributed strategy: each individual thread decides whether to proceed or postpone the method call while executing `pre_call()`. These decisions are based on analyzing the type-state automaton specification, which shows that executing `o.destroy()` will lead to a sink state, from which no other method calls can be executed (see Figure 3.3). That is, method `o.destroy()` should not be allowed to execute while the program is still in states (a_1, b_1), (a_2, b_1), or (a_3, b_1); otherwise, based on the control flow information of the program, we know that the pending requests in thread T_1 for executing `o.init()`, `o.start()`, and `o.stop()` will cause a violation.

To sum up, our strategy in general is to avoid moving the automaton to a state from which

violations become inevitable. The underlying theory for our framework is model checking, where the control flow graph serves as a finite state abstraction of the client program. The inevitably faulty states in the composed system of the program and the automaton can be characterized by a temporal logic formula $\mathsf{AF}\Psi_{bad}$. Here, path qualifier A means *on all paths*, temporal operator F means *finally*, and predicate Ψ_{bad} represents the set of illegal states as defined by the type-state automaton.

When the state transition system M is finite, formula $\mathsf{AF}\Psi_{bad}$ is decidable [22, 51]. In this case, our new method can be optimized with three criteria in mind: *safety*, *effectiveness*, and *cost*. Safety means that the runtime mitigation actions do not introduce erroneous thread interleavings that do not already exist in the original program. Effectiveness means that all invalid interleavings should be removed and all valid interleavings should be retained. Cost means that the mitigation actions added to `pre_call()` should have the smallest possible runtime overhead.

3.2 The Scope

In this section, we define the type of bugs targeted by our method and outline the main technical challenges.

The Scope. Concurrency bugs can be divided into two groups based on whether they are caused by (1) the synchronization being too loose or (2) the synchronization being too tight. When the synchronization is too loose, there will be unwanted interleavings, causing data-races, atomicity violations, order violations, etc. When the synchronization is too tight, some interleavings will be prohibited, causing deadlocks, live-locks, or performance bugs. Our new method is based on tightening up the synchronization by analyzing the interaction between the client and the type-state automaton of the library. As such, it is designed to prevent bugs caused by the thread synchronization being too loose (Type 1). Bugs in the other group (Type 2) will not be handled by our technique.

It is worth pointing out that not all object behaviors in C++/Java can be expressed by using *type-state automaton* [125]. Objects with *blocking* methods (such as locking operations) are obvious examples. Consider `mutex_lock`, for instance, it is not clear what the type-state automaton specification should be. The straightforward solution, where the `o.lock()` and `o.unlock()` method calls are required to be strictly alternating, is incorrect because *which thread executes a method* is also crucial to deciding whether a call sequence is legal. Unfortunately, the notion of threads is not part of the classic definition of type-state automaton [125] – this is a well known theoretical limitation of type-state automaton.

Our work is not about lifting this theoretical limitation. Instead, our work focuses on leveraging the current form of type-state automaton – which is widely used in object-oriented development practice – to mitigate bugs caused by incorrect use of synchronizations external to the methods. Therefore, we assume that all the concurrency control operations are imposed by the client program, and all methods of the type-state automaton itself are *non-blocking*, e.g., there is no internal locking operation. All locking operations are performed by the client program outside the methods.

The Technical Challenges. One of the main challenges in runtime failure mitigation is to make sure that the mitigation actions are safe. That is, the additional actions should not introduce new erroneous interleavings, e.g., deadlocks resulting from the thread synchronization being too loose to the thread synchronization being too tight. At the same time, they should be able to steer the program to a failure-free interleaving whenever such interleaving exists. This is actually difficult to do because, as we have mentioned before, most of the existing methods do not guarantee that they would not introduce new deadlocks. We can avoid introducing new bugs because of two reasons. First, we have the type-state automaton, which serves as a clearly defined goal for perturbing the thread interaction. Second, we propose a rigorous theoretical framework for analyzing and deriving the runtime mitigation strategies, based on the theory of model checking.

Our example in Figure 3.1 may give the impression that runtime prevention of concurrency

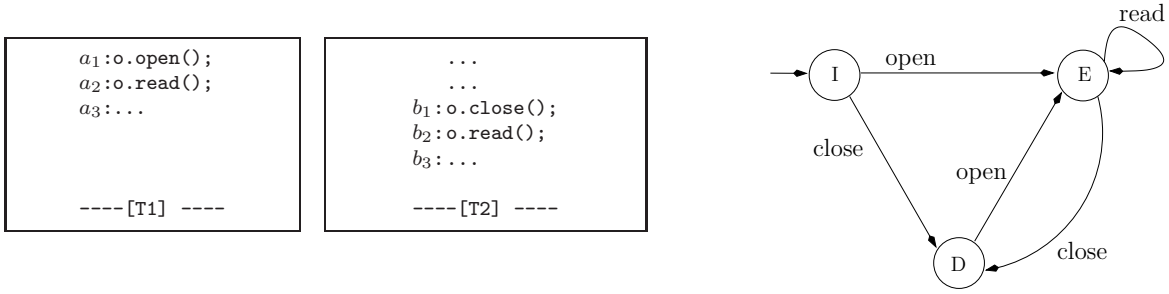


Figure 3.5: A client program and the type-state automaton. All valid thread interleavings must start with $b_1, a_2, \ldots$

related type-state errors is easy. However, this is not the case. Consider Figure 3.5, where the automaton for `file_reader` as defined in the Boost C++ library says that `read()` can execute after `open()` and `read()`, but not immediately after `close()`. At first glance, it may be tempting to design a strategy as follows:

- in automaton state I, delay `read()`;
- in automaton state E, delay `open()`;
- in automaton state D, delay `read()` and `close()`.

However, this strategy is not sufficient to suppress the violation. To suppress the violation, starting with the initial state (a_1, b_1), we should execute thread T_2's `close()` while delaying thread T_1's `open()`. However, such decision cannot be made based on the above strategy. Indeed, it is not immediately clear why one should favor T_2's `close()` over T_1's `open()`, since both are allowed to execute while the automaton is in State I.

This example highlights one challenge in designing the runtime strategies. Rather than coming up with *ad hoc* strategies as in existing methods and then trying to justify why they work, it would be more appealing to establish a unified theoretical framework within which the effectiveness and cost of various strategies can be analyzed. This is what we set out to do in this chapter – it is a main difference between our new method and the existing methods.

Another main challenge is to address the competing requirement of reducing the runtime overhead. In practice, finding the failure-free interleaving whenever it exists may require full

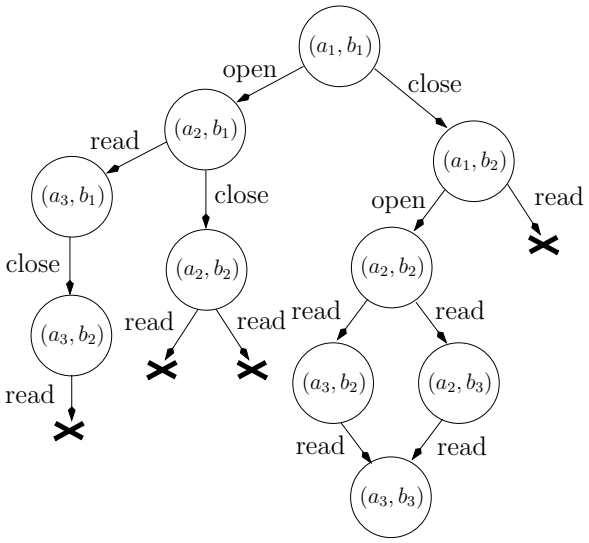


Figure 3.6: State transitions for the example in Figure 3.5.

observability of the client program's state, and therefore is prohibitively expensive. Therefore, we need to design the mitigation strategies under the assumption that we only have the control flow information of the client program, which is an over-approximation that we can obtain through static program analysis.

3.3 Preliminaries

In this section, we formally define the type-state automaton and the over-approximate state transition system of the client program.

3.3.1 Type-State Automaton

A type-state automaton is a specification of the temporal properties of the API usage rules associated with an object. In object-oriented programming languages, an object is a container for data of a certain type, providing a set of methods as the only way to manipulate the data. Typical API usage rules are as follows: if the object is in a certain state before calling the method, it will move to another state after the call returns, and the call will

return a certain value or throw an exception.

A type-state automaton is an edge-labeled state transition graph $A = \langle S, T, s_0, S_{bad}, L \rangle$, where S is a set of states, $T \subseteq S \times S$ is the transition relation, $s_0 \in S$ is the initial state, S_{bad} is the set of illegal states, and L is the labeling function. Each transition $t \in T$ is labeled with a set $L(t)$ of method calls. A valid method call sequence corresponds to a path in the automaton that starts at s_0 and ends at a state $s \notin S_{bad}$. For example, when the stack is empty, executing `pop()` would force the automaton to a state in S_{bad}. States in S_{bad} are terminal: once the automaton enters such a state, it will remain there forever. Therefore, a type-state automaton specifies a set of temporal safety properties.

In the original definition of type-state automaton [125], states do not have labels. In order to represents objects such as stacks and queues, which may have a large or even infinite number of states, we use a generalized version (e.g., [144]) where a state may be labeled with a set of Boolean predicates – transition to the state is allowed only if all these predicates evaluate to true. It is worth pointing out that, even with this extension, the automaton remains an abstraction of the object, and there exist other extensions such as parameterized and dependent type-states (c.f. [24]), which make the automaton even more expressive. However, we have decided not to explore these extensions due to performance concerns.

3.3.2 Concurrent Program

A concurrent program has a set of *shared* objects SV and a set of threads $T_1 \ldots T_m$. Each thread T_i, where $1 \leq i \leq m$, is a sequential program with a set of *local* objects. Let st be a statement. An execution instance of st is called an *event*, denoted $e = \langle tid, l, st, l' \rangle$, where tid is the thread index, l and l' are the thread program locations before and after executing st. We model each thread T_i as a state transition system M_i. The transition system of the program, denoted $M = M_1 \times M_2 \times \ldots \times M_m$, is constructed using the standard interleaved composition [22, 51]. Let $M = \langle Q, E, q_0, L \rangle$, where Q is the set of global states, E is the set of transitions, q_0 is the initial state, and L is the labeling function. Each state $q \in Q$

is a tuple of thread program states. Each transition $e \in E$ is an event from one of the m threads. The labeling function L maps a method call event to the method name, and maps a non-method call event to the empty name ϵ.

An execution trace of M is a sequence $q_0 \xrightarrow{e_1} q_1 \xrightarrow{\cdots} q_n$, where $q_0, \ldots, q_n$ are the states and $e_1, \ldots, e_n$ are the events. A method call event from thread T_i may have one of the following types:

- $o.m()$ for calling method $m()$ of shared object o;
- $fork(j)$ for creating child thread T_j, where $j \neq i$;
- $join(j)$ for joining back thread T_j, where $j \neq i$.
- $lock(lk)$ for acquiring lock lk;
- $unlock(lk)$ for releasing lock lk;
- $signal(cv)$ for setting signal on condition variable cv;
- $wait(cv)$ for receiving signal on condition variable cv;

Here, $o.m()$ represents a method call of the shared object, while the remaining ones are thread synchronization operations.

The reason why we model thread synchronization events is that we need to decide which threads are blocking at any time during the execution. The information is important to avoid introducing artificial deadlocks by the online mitigation algorithm. In defining the above events, we have assumed that the client program uses thread creation and join, mutex locks, and condition variables for synchronization operations. Locks and condition variables are the most widely used primitives in mainstream platforms such as PThreads, Java, and C#. If other blocking synchronizations are used, they also need to be modeled as events. By doing this, we will be able to know, at each moment during the program execution, which threads are in the blocking mode (*disabled*) and which threads are in the execution mode (*enabled*).

3.3.3 Abstraction and Limited Observability

In practice, the precise model M of the client program can be extremely large or even have infinitely many states. Since our focus is on enforcing API usage rules, we consider a finite state abstraction of M, denoted $\widehat{M}$. In this abstraction, we do not consider the thread-local events. Furthermore, we collapse a set of concrete states at each program location to form an abstract state.

More formally, we define a control flow abstraction of the program, denoted $\widehat{M} = \langle \widehat{Q}, \widehat{E}, \widehat{q}_0, \widehat{L} \rangle$, where $\widehat{Q}$ is a set of control locations, $\widehat{E}$ is a set of transitions, $\widehat{q}_0$ is the initial state, and $\widehat{L}$ is the new labeling function. We make sure that $\widehat{M}$ is an over-approximation of M in that every path in M is also a path in $\widehat{M}$, although $\widehat{M}$ may have spurious paths that do not exist in M.

For example, if each individual thread is modeled by its control flow graph, $\widehat{M}$ would be the interleaved composition of all these control flow graphs. In this case, the abstract state is a *global control state*, denoted $\widehat{q} = \langle l_1, \ldots, l_m \rangle$, where each l_i is a location in thread T_i. An *abstract* state in this model would encompass all the concrete program states that share the same set of thread locations. Abstractions such as this, together with the labeling function $\widehat{L}$, provide only limited observability of the client program but they allow the runtime analysis and mitigation problem to be decidable.

3.4 Runtime Failure Prevention

In this section, we focus on developing a theoretical framework for analyzing the runtime failure mitigation problem. However, methods presented here are not meant to be directly implemented in practice. Instead, they represent what we would like to achieve in the ideal case. In the next section, we shall develop a practically efficient implementation scheme, by carefully over-approximating the ideal methods presented in this section.

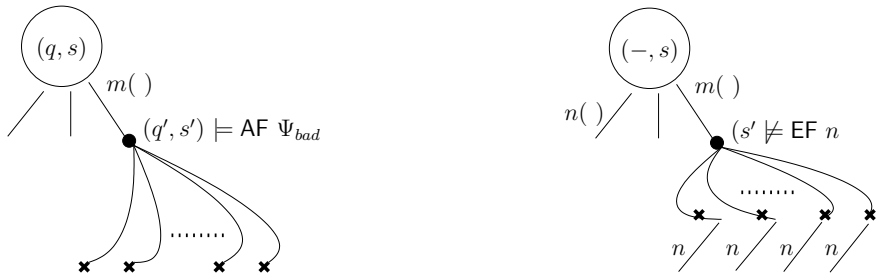
Here, we assume that both the automaton A and the client program M are available, and we show that our method guarantees to satisfy both the *safety* and the *effectiveness* criteria. We leave the optimization of runtime *cost* – which assumes the availability of $\widehat{M}$ instead of M – to the subsequent sections.

3.4.1 The Theory

Recall that our prevention method is based on imposing additional constraints on the order of method calls in the client program, to suppress erroneous interleavings. Furthermore, the runtime control is implemented as a distributed strategy where each thread, upon receiving the method call request, decides whether to postpone its execution. Although delaying method invocations may seem to be an approach that hurts performance, it is actually unavoidable because the delay is dictated by the desired client program behavior as specified by the type-state automaton specification. In other words, the delay is a necessary part of the correct program behaviors. Therefore, it is only a matter of enforcing the delay through the synchronization mechanism.

The theoretical foundation for our runtime analysis and mitigation is a *state space exploration* of the composed system $(M \times A)$: That is, our runtime actions depend on analyzing the type-state automaton A and its interaction with the client program M. At the high level, our strategy is as follows: at each step of the program execution, we check whether executing a certain method call m will inevitably lead to a type-state violation. If the answer to this question is yes, we delay the execution of that method.

More formally, let the transition system $G = M \times A$ be the synchronous composition of the client program M and the type-state automaton A using the standard notion in state space exploration [22, 51]. Each node in G is a pair of states in M and A of the form (q, s), where $q \in Q$ and $s \in S$. Each edge $(q, s) \xrightarrow{e} (q', s')$ corresponds to the simultaneous execution of transition $q \xrightarrow{e} q'$ in M and transition $s \xrightarrow{e} s'$ in A. The initial state is (q_0, s_0), where q_0 is the initial program state and s_0 is the initial automaton state. The set of bad states is



(a) theoretical method

(b) practical method

Figure 3.7: Illustrating the correctness proof of our runtime analysis and error mitigation strategy.

defined as $\{(q, s) \mid s \in S_{bad}\}$.

In the transition system $G = M \times A$, we decide at any state (q, s), whether transition $(q, s) \xrightarrow{m} (q', s')$ should be executed by checking whether (q', s') satisfies the temporal logic property $\mathsf{AF}\ \Psi_{bad}$. Here, path quantifier A means *on all paths*, temporal operator F means *finally*, and predicate Ψ_{bad} means a bad state in S_{bad} has been reached.

Specifically, if state (q', s') satisfies $\mathsf{AF}\ \Psi_{bad}$, denoted $(q', s') \models \mathsf{AF}\ \Psi_{bad}$, then from automaton state s', the type-state violation becomes inevitable. This is illustrated in Figure 3.7 (a) – we say that transition $(q, s) \xrightarrow{m} (q', s')$ will lead to a type-state violation. Notice that a special case is when state s' itself is a bad state. The more general case is when all paths originated from s' lead to a bad state. Since our goal is to avoid such concurrency related type-state violations, whenever executing method m in state (q, s) leads to a state $(q', s') \models \mathsf{AF}\ \Psi_{bad}$, we postpone the execution of method m, thereby avoiding the subsequent state (q', s').

The pseudocode is shown in Algorithm 4. Recall that we have inserted an `pre_call(`m`)` before every call to method m of the shared object under inspection. Inside this procedure, we first check if the current thread T is the only enabled thread. If the answer is yes, there is no scheduling nondeterminism at this time, and we have no choice but to execute m (if executing m inevitably causes a violation, the violation is already unavoidable). If T is not the only enabled thread, we call `must_delay(`m`)` to check whether executing m leads to a state $(q', s') \models \mathsf{AF}\ \Psi_{bad}$. If this is the case, `must_delay(`m`)` returns `true`; otherwise, it returns

false.

Whenever `must_delay(m)` remains true, thread T_1 waits inside the while-loop until `must_delay(`m`)` eventually returns false. Inside the while-loop, we call `usleep()` to release the CPU temporarily so that it can execute other threads. Meanwhile, another thread may execute `pre_call(`n`)` and change the state from (q, s) to (q'', s''). This can make `must_delay(`m`)` returns true, thereby allowing thread T to break out of the loop. At this time, thread T will advance the global state based on $(q'', s'') \xrightarrow{m} (q''', s''')$ and then return. After that, method call m will be executed.

Algorithm 4 (Theoretical) Deciding whether to delay method m.

```
 1: pre_call ( object o, method m ) {
 2:     Let T be this thread;
 3:     while (T is not the only enabled thread) {
 4:         while ( must_delay( o, m ) ) {
 5:             usleep( usec );
 6:         }
 7:     }
 8:     update_state( o, m );
 9: }
10: must_delay( object o, method m ) {
11:     Let (q, s) be the current state of M × A_o;
12:     Let (q', s') be the state after executing m in state (q, s);
13:     if ( (q', s') ⊨ AF Ψ_bad )     return true;
14:     return false;
15: }
```

The spinning of `pre_call()` at Lines 4-5 is implemented by calling `usleep()`, which releases the CPU temporarily to execute other threads. We set an increasingly longer delay time in `usleep` by using the popular *exponential back-off* strategy [49]. During our experiments, the delay time starts from MIN=1ms to MAX=1s.

Example 1 *Consider the client program in Figure 3.1 and the state transition graph in Figure 3.3. When the program is at state* (a_1, b_1)*, we have to delay* T_2*'s request to execute* `o.destroy()`*. The reason is that, executing* `o.destroy()` *would move the program from* (a_1, b_1) *to* (a_1, b_2)*. However,* $(a_1, b_2) \models \mathsf{AF}\Psi_{bad}$ *meaning that the type-state violation becomes inevitable starting from* (a_1, b_2)*. In contrast, we allow thread* T_1 *to execute* `o.init()` *because*

it would move the program to state (a_2, b_1)*, which does not satisfy* $\mathsf{AF}\Psi_{bad}$ *due to the existence of the left-most path in Figure 3.3.*

Example 2 *Consider the client program in Figure 3.5 and the state transition graph in Figure 3.6. When the program is in state* (a_1, b_1)*, we have to delay thread* T_1*'s request to execute* `o.open()`*, because the next state* $(a_2, b_1) \models \mathsf{AF}\Psi_{bad}$*, that is, all paths from* (a_2, b_1) *eventually lead to a type-state error. In contrast, we allow thread* T_2 *to execute* `o.close()`*, because the next state* (a_1, b_2) *does not satisfy* $\mathsf{AF}\Psi_{bad}$ *due to the existence of paths that end at state* (a_3, b_3) *in Figure 3.6.*

To determine whether a running thread is *enabled*, which is required by Line 3 of Algorithm 4, we monitor the thread synchronization events in addition to the method calls of the shared object.

- An currently enabled thread T becomes disabled when it executes method m such that (i) m is `lock(`lk`)` but lock lk is held by another thread, (ii) m is `wait(`cv`)` but the condition variable cv has not been set by a remote thread, or (iii) m is a thread `join(`j`)` but child thread T_j has not terminated.
- Similarly, a disabled thread T becomes enabled when another thread executes method n such that (i) n is `unlock(`lk`)` that releases the lock, (ii) n is either `signal(`cv`)` or `broadcast(`cv`)` that sets the condition variable cv waited by thread T, or (iii) n is the termination event of child thread waited by thread T.

3.4.2 Proof of Correctness

Now, we prove that our method satisfies both the *safety* property and the *effectiveness* property defined as follows.

Theorem 1 (Safety) *The runtime mitigation method in Algorithm 4 never introduces new concurrency bugs that do not exist in the original client program.*

In other words, we guarantee that the runtime actions will *do no harm*. We provide the proof sketch as follows: In this algorithm, the only impact that these actions have on the client program is the delay of some method calls. The result of such delay is that certain thread interleavings, which are allowed by the original program, are no longer allowed. However, adding delay to the client program will not introduce new interleavings to the program. Since all the possible thread interleavings of the new program are also valid interleavings of the original program, our method does not introduce any new program behavior (no new concurrency bug). At the same time, our new method will not introduce artificial deadlocks, because of the implicit check for artificial deadlocks in Line 3 of our Algorithm 4: If thread T is the only enabled thread at that time, and it will not be delayed by our method. □

Theorem 2 (Effectiveness) *When the precise state transition system M of the client program is available and M has a finite number of states, the method in Algorithm 4 can guarantee to steer the program to a failure-free interleaving, as long as such interleaving exists.*

We provide the proof sketch as follows: Under the assumption that M is precise and is finite in size, the evaluation of $(q', s') \models \mathsf{AF}\ \Psi_{bad}$ is decidable [22, 51]. This will directly establish the effectiveness of our method. At each node in the composed graph $M \times A$, we can avoid executing method m if it leads to state $(q', s') \models \mathsf{AF}\ \Psi_{bad}$. This is illustrated pictorially by Figure 3.7 (a). As long as the initial state $(q_0, s_0) \not\models \mathsf{AF}\ \Psi_{bad}$, meaning that there exists a failure-free interleaving, our method will ensure $\mathsf{AF}\ \Psi_{bad}$ is false in all subsequent states of the chosen execution trace. □

Our proof for Theorem 2 contains an important message. When the client M has infinitely many states, it is impossible to guarantee the effectiveness property. The reason is that checking temporal logic formulas such as $\mathsf{AF}\ \Psi_{bad}$ in an infinite-state system is undecidable. Even if M is a finite-state system, constructing and analyzing this potentially large graph M can become prohibitively expensive. Therefore, we substitute M by a finite-state abstraction as defined in Section 3.3, where we use the control flow graph of each thread to

over-approximate its state transition system. In this case, the safety property can still be guaranteed.

Let $\widehat{G} = \widehat{M} \times A$. Instead of evaluating $(\widehat{q}', s') \models \mathsf{AF}\ \Psi_{bad}$ in G, we now choose to evaluate $(\widehat{q}', s') \models \mathsf{AF}\ \Psi_{bad}$ in $\widehat{G}$. Since $\widehat{G}$ is an over-approximation of G, it contains all valid paths of G, and possibly some bogus paths. Therefore, by the definition of AF, which means *on all paths eventually*, we have

$$(\widehat{q}', s') \models \mathsf{AF}\ \Psi_{bad} \quad \text{implies} \quad (q', s') \models \mathsf{AF}\ \Psi_{bad}\ ,$$

meaning that if all paths from $(\widehat{q}', s')$ in $\widehat{G}$ end at a bad state, then all paths from (q', s') in G end at a bad state. But the reverse is not true since some added paths in $\widehat{G}$ may end at a good state.

Based on the above analysis, we conclude that, by using abstract model $\widehat{M}$ to replace M, and using $(\widehat{q}', s') \models \mathsf{AF}\ \Psi_{bad}$ to replace the condition in Line 13 of Algorithm 1, we will still be able to retain the *safety* guarantee. However, depending on the accuracy of $\widehat{M}$, we may not always be able to satisfy the effectiveness guarantee, because it can only provide us with limited observability of the program. The important message is that, the loss of *effectiveness* guarantee is theoretically unavoidable.

3.5 Algorithm

In this section, we present a practically efficient instantiation of the method introduced in the previous section. It is defined by a set of easily checkable sufficient conditions whose validity implies $(q', s') \models \mathsf{AF}\ \Psi_{bad}$. They allow us to avoid checking the precise client model M, or even the finite-state abstraction $\widehat{M}$. In fact, this new algorithm relies mainly on the type-state automaton A, while requiring little information about the client program.

3.5.1 Rules Based on the Automaton Only

First, we present a sufficient condition for $(q', s') \models \mathsf{AF}\ \Psi_{bad}$ that can be evaluated by analyzing the automaton A only. The condition is meant to replace the check in Line 13 of Algorithm 4.

Similar to the previous section, let s be the current state of the automaton A, m be the object method to be executed in thread T, and s' be the next state of the automaton A if we execute m in state s. Instead of checking $(q', s') \models \mathsf{AF}\Psi_{bad}$, we check if at least one of the following conditions holds:

- **Rule 1:** If $s' \in S_{bad}$, meaning that a violation will happen in s', we must postpone the execution of method m.
- **Rule 2:** If there exists a future request to execute method n in state s' by another thread T', but in the automaton A, there does not exist any edge labeled with n that is reachable from state s', we must postpone the execution of method m.

These conditions are easily checkable because they depend only on the current state of M and future method calls of the automaton A. None of them requires the state transition information of M or $\widehat{M}$. Indeed, Rule 1 involves the current and next automaton states s and s' only. Rule 2 involves, in addition to s and s', also the future method calls (n) of other threads, and the reachability of edges labeled with n in automaton A. Therefore, Rules 1 and 2 are sufficient conditions for $(q', s') \models \mathsf{AF}\ \Psi_{bad}$:

- In Rule 1, if s' is a bad state, $\mathsf{AF}\ \Psi_{bad}$ is already satisfied.
- In Rule 2, when both n and m are enabled at state $(-, s)$ and we choose to execute m, as illustrated in Figure 3.7 (b), n must appear in all execution paths starting with $(-, s')$. However, in the automaton A, if edges labeled with n are not reachable from state s', then all paths of the composed system starting with $(-, s')$ are illegal. This means $(-, s') \models \mathsf{AF}\ \Psi_{bad}$.

The pseudo code of our new method is shown in Algorithm 5. In contrast to the original algorithm, We replace `update_state()` with `update_automaton-state()`, since states of the client program M are no longer used. By default, `must_delay()` returns false, meaning that call to this method will not be delayed. Line 13 implements Rule 1. Lines 15-18 implement Rule 2. If any of the two sufficient conditions holds, the procedure returns true, meaning that call to this method must be delayed, until other threads execute some method that can change the current automaton state s, and make `must_delay()` return false.

Algorithm 5 (Practical) Deciding whether to delay method m.

```
 1: pre_call ( object o, method m ) {
 2:     Let T be this thread;
 3:     while (T is not the only enabled thread) {
 4:         while ( must_delay( o, m ) ) {
 5:             usleep( usec );
 6:         }
 7:     }
 8:     update_automaton-state( o, m );
 9: }
10: must_delay( object o, method m ) {
11:     Let s' be the automaton state after executing m in state s;
12:     // Rule 1.
13:     if ( s' ∈ S_bad ) return true;
14:     // Rule 2.
15:     for each ( method n in another thread T' ) {
16:         if ( n is not reachable from s' in automaton A )
17:             return true;
18:     }
19:     // Rule 3.
20:     for each ( pattern collected from the client program ) {
21:         if ( pattern is not reachable from s' in automaton A )
22:             return true;
23:     }
24:     return false;
25: }
```

It is worth pointing out that Rules 1 and 2 are sufficient but not necessary conditions. They can guarantee the safety property but may not be strong enough to guarantee the *effectiveness* property. However, it does not mean that the *effectiveness* property cannot be achieved in most of the practical settings. Indeed, we have found that these two rules are already powerful enough to prevent most of the concurrency related type-state violations encountered in our benchmarks. We will present our experimental results in Section 3.7.

3.5.2 Rules Based on Limited Observability of Client Program

When additional information about the client program is available, we can leverage the information to improve the effectiveness of the algorithm. For example, the concurrency error in Fig 3.5 cannot be avoided by applying Rules 1 and 2 only, or any rule that is based exclusively on the automaton. This is because both T_1's `open()` and T_2's `close()` can pass the checks of Rules 1 and 2. Therefore, `must_delay(m)` will return false in both threads. However, the failure-free interleaving requires that we postpone `open()` in thread T_1, while allowing the execution of `close()` in thread T_2.

If we can somehow look ahead a few steps in the execution of each thread, e.g., by leveraging the control flow information that we obtain through static program analysis, as shown in Figure 3.2. Our runtime mitigation algorithm will be able to perform better. We now present an extension to incorporate such static information into our runtime analysis. The result is an additional rule, which can successfully handle the example in Figure 3.5.

The new rule is based on identifying call sequences that may appear in the client program, denoted $pattern(m, n, \ldots)$.

- **Rule 3.** If there exists a pending method call sequence characterized by $pattern(m, n, \ldots)$ in the client program, but in automaton A, there exists no valid path that starts with state s' and leads to a sequence that matches $pattern(m, n, \ldots)$, we must postpone the execution of method m.

Example 3 *Consider the client program in Figure 3.5. By analyzing the client program code, we know that the* `close()` *method call is always followed by a call to* `read()` *from the same thread. Therefore, any interleaving must have* `close(),...,read()`*. Furthermore, we know that* a_1 *has the only call to* `open()`*. If we were to execute* a_1 *first, after that, there would be no call to enable and the pattern becomes* `close[!open]*read`*. However, this pattern is not reachable from the current automaton state. Based on Rule 3, therefore,* `must_delay()` *will have to return* true *for* T_1*'s request to execute* `open()`*, but return* false *for* T_2*'s request*

to `close()`.

We use Rule 3 in Algorithm 5 in the same way as we use Rule 2, except that the patterns must be collected as *a priori* from the client program before they are checked against the automaton A at run time. These patterns are collected from the client program through a conservative static analysis of the control flow graphs of the individual threads, which will be explained in Section 3.6.

3.6 Static Analysis

The static analysis as required in Figure 3.2 has to be conservative in that it must over-approximate the control flow of the program. However, it does not need to be precise for our runtime mitigation to be effective. To avoid the well-known interleaving explosion problem, we carry out this static analysis on each individual thread in a *thread-local* fashion.

3.6.1 Collecting Future Events

Our static analysis computes the following information:

- For each method call m in a thread, compute the set of methods $\{n\}$ such that n is the next method that *may* be executed immediately after m by the same thread. We represent m and $\{n\}$ as a key-value pair in a table `(method` $\rightarrow$ `nextMethods)`.
- For each method call m in a thread, compute the set of methods $\{n'\}$ such that n' is a future method that *may* be executed some time after m by the same thread. We represent m and $\{n'\}$ as a key-value pair in a table `(method` $\rightarrow$ `futureMethods)`.

These tables are statically computed as *a priori* and then made available at runtime. Our runtime mitigation algorithm will leverage them for evaluating Rule 3 in Algorithm 5, thereby improving the effectiveness of our method in suppressing concurrency errors.

There can be many different ways of implementing such static program analysis, as long as the analysis result is conservative. That is, if event m *may* appear before event n' in some actual execution of the program, then n' *must* be included in the future (next) event table of m. A nice feature of our method is that the next and future event tables do not need to be precise for the runtime mitigation to be effective. Indeed, even without the next/future event tables, our runtime mitigation method still works correctly: it is able to ensure safety and remains effective in most cases (e.g., as in Rule 1). However, the next/future event tables, whenever they are available, can help cut down the delay introduced by Rule 2 and can be used to implement Rule 3.

3.6.2 Reducing Runtime Overhead

Now, we show that the statically computed next/future event tables may also be used to reduce the runtime overhead of applying Rule 2. To understand why this is the case, consider the following example. While evaluating Rule 2 for method m in thread T_2, we need to know the next method calls of all other concurrent threads, including method n in thread T_1 in the example below. However, without the pre-computed event tables, some of the information would not have been available.

```
--------[T1]--------    --------[T2]--------
  pre_call(n1)
  n1();
  ...                     must_delay(m)
  pre_call(n2)
  n2();
--------------------    --------------------
```

In this example, the call to method m in thread T_2 is executed between methods n_1 and n_2. When `must_delay`(m) is executed, thread T_1 may still be executing the code before `pre_call(`n_2`)`. Since it has not yet entered `pre_call(`n_2`)`, it has not updated the data

shared with thread T_2. Therefore, while T_2 evaluates `must_delay`(m), it will not be able to know what the next method call in thread T_1 is. This poses a problem for checking Rule 2.

If the event tables as described in Section 3.6.1 are not available, we have two choices:

- We may allow `must_delay`(m) to consider only the arrived method calls from other threads. If a thread is still *in-flight*, such as T_1 in the above example, we simply skip it while executing Lines 6-10 in Algorithm 5. This approach is good for performance, but may reduce the effectiveness in suppressing bugs. For example, in Figure 3.1, we may not be able to delay T_2's call to `destroy()` if T_1 is *in-flight*.
- We may require `must_delay`(m) to wait for all threads to arrive and report the method calls before making a decision on whether to delay the method m. This approach will retain the effectiveness of our method in suppressing bugs, but may hurt performance.

None of the two choices is desirable.

With the help of the pre-computed next/future event tables, we will be able to overcome this dilemma. For each method m executed by thread T_i, return the immediate next method n that *may be called by the same thread*, we can leverage the successor event information to help remove the limitations associated with *in-flight* threads and pending method calls. We no longer need to compromise on the effectiveness by ignoring the next method calls from *in-flight* threads, or compromise on the performance.

3.7 Experiments

We have implemented the proposed method in a tool based on the LLVM compiler framework to handle arbitrary C/C++ applications based on the PThreads. We use LLVM to conduct static program analysis and then inject runtime analysis and control code to the client program, so that the relevant method calls of shared objects are monitored at run time. More specifically, we insert the `pre_call()` routine before every thread synchronization

routine and every blocking system call to determine *which threads are enabled* during the execution. We also insert the `pre_call()` routine before every call to methods of the shared objects in order to control their execution order. The type-state automaton, which provides the object type and monitored methods, are manually edited into configuration files, which in turn serve as the input of our tool.

During our experiments, we consider three research questions:

- How well can it suppress concurrency related type-state violations in real-world applications?
- How well can it control the performance overhead of the runtime mitigation actions?
- How well does it scale on applications with large code base and many concurrent threads?

Evaluation Methodology. We have conducted experiments on a set of open-source C/C++ applications on the Linux platform to evaluate our new method. The characteristics of all benchmark examples are summarized in Table 3.1.

The first two benchmark examples are full-sized open-source applications with known bugs and the corresponding test cases to reproduce them. `Memcached-1.4.4` is a high-performance, distributed memory object caching system. The bug happens when two clients concurrently increment or decrement certain cached data: the in-place increment/decrement is not atomic, causing some updates to be lost [158]. `Transmission-1.42` is a multithread BitTorrent download client. The bug happens when reading a shared variable should occur after initialization, but may occur before the initialization due to the lack of synchronization, causing an assertion failure.

The next nine benchmark examples are unit-level test programs for the popular `Boost` C++ libraries, which are portable source libraries used by many commercial and open-source applications. We have carefully studied the APIs of some of the most popular packages such as `timer` and `basic_file` and created the corresponding type-state automata.

Table 3.1: The evaluated applications and the descriptions of the type-state violations.

Appl. Name	LoC	Class Name	Bug Description
Transmission-1.42	90k	`t->peerMgr`	Reading of `h->bandwidth` should occur after initialization, but may be executed before initialization due to lack of synchronization, causing an assertion failure.
Memcached-1.4.4, bug id:127	62k	`key`	The bug happens when two clients concurrently increment or decrement certain cached data. The in-place incr/decr is not atomic, causing some updates being lost.
Boost-1.54.0:file reader	48k	`basic_file`	If one thread called **close** earlier, another thread that tries to call **read** would not be able to read the correct content from the target file.
Boost-1.54.0:file writer	48k	`basic_file`	If one thread called **close** earlier, another thread that tries to call **write** would not be able to write the content correctly to the target file.
Boost-1.54.0:timer	27k	`cpu_timer`	Supposedly a timer should be resumed when it's stopped earlier, if one thread called **resume** before another thread calls **stop**, the timer would still be stopped thus not functioning.
Boost-1.54.0:object_pool	54k	`pool`	If one thread called the de-constructor ∼**pool** earlier, another thread that tries to call **release_memory** would no be able to get the correct return value.
Boost-1.54.0:move	45k	`file_descriptor`	After one thread called **boost::move**, the new pointer gets the value of the old pointer and the old pointer becomes invalid, another thread that tries to call **empty** of the old pointer would not be able to get the correct return value.
Boost-1.54.0:unorderd_map	53k	`unorderd_map`	If one thread called the de-constructor ∼**unordered_map** earlier, another thread that tries to call **operator**[ ] would not be able to get the correct return value.
Boost-1.54.0:unorderd_set	53k	`unorderd_set`	If one thread called the de-constructor ∼**unordered_set** earlier, another thread that tries to call **size** would not be able to get the correct return value.
Boost-1.54.0:any	27k	`any`	If one thread called the de-constructor ∼**any** earlier, another thread that tries to call the function **type** would cause the program to crash.
Boost-1.54.0:priority_queue	54k	`fibonacci_heap`	If one thread tries to call **pop** when the priority_queue is empty, it would cause the program to crash.

In all cases, the automaton is specified by the user. The automaton size is small (less than 10 states). In addition, for each benchmark example, the user provides a buggy test program and a bug-free test program. Together, they serve as input of our tool during the experimental evaluation.

Effectiveness. We have evaluated the effectiveness of our method in suppressing type-state violations. Table 3.2 shows the results. Columns 1-2 show the statistics of the benchmarks, including the name and the number of threads encountered during the test run. Columns 3-4 show the results of running the original program, where we show whether the violation has

Table 3.2: The effectiveness for runtime failure mitigation.

Test Program		Original		Prevention	
name	threads	error	time	error	time
Transmission	2	crash	(5.5 s)	fixed	12.6 s
Memcached-127	4	crash	(10.1 s)	fixed	20.1 s
Boost:file_read	2	fail	(15.1 s)	fixed	15.5 s
Boost:file_write	2	fail	(11.4 s)	fixed	12.7 s
Boost:timer	2	fail	(9.0 s)	fixed	9.3 s
Boost:object_pool	2	fail	(15.2 s)	fixed	15.6 s
Boost:move	2	fail	(10.0 s)	fixed	10.4 s
Boost:unorderd_map	2	fail	(10.2 s)	fixed	10.8 s
Boost:unorderd_set	2	fail	(11.4 s)	fixed	12.6 s
Boost:any	2	crash	(¡0.1 s)	fixed	8.6 s
Boost:priority_queue	2	crash	(¡0.1 s)	fixed	9.4 s

occurred and the execution time (before the program crashes). Columns 5-6 show the results of running the same program but with runtime prevention. Again, we show whether the bug has occurred and the execution time. These experiments were conducted on a workstation with 2.7 GHz Intel Core i7 (with four logical processors) and 8GB memory.

In all test cases, our method was effective in suppressing these concurrency related type-state violations. We have also run the fixed program multiple times, to see if the fixes are accidental or consistent. Our results show that the fixes are consistent across different runs: after applying our new method, the known concurrency errors never show up again.

Performance. We have evaluated the performance overhead of our method using the same set of applications but with a different set of test programs (non-buggy ones). The reason for using a different set of test programs is that, the set of test programs used in Table 3.2 are not suitable for comparing performance. Without mitigation, the test programs might crash in the middle of the execution. For the open-source applications, each buggy test case also has a failure-free test case. For the `Boost` examples, we also have failure-free versions of all the test programs. For these test cases, both the original program and the program with prevention can complete. The additional computation required by our runtime prevention algorithm adds nothing but pure runtime overhead.

Figure 3.8 shows the results. The x-axis are the benchmarks. The y-axis are the runtime

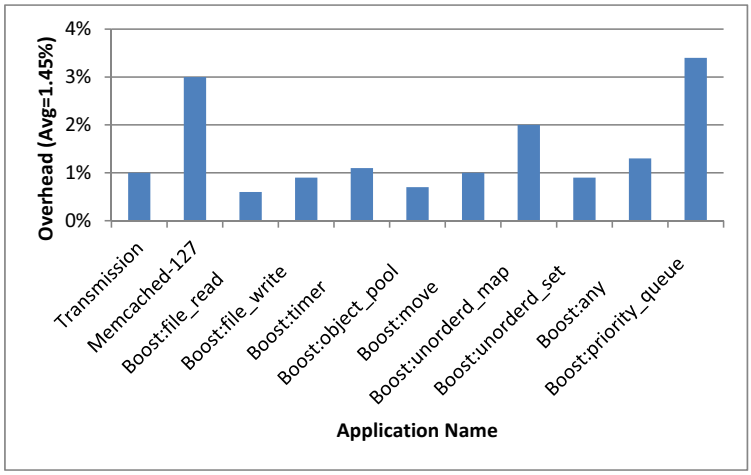


Figure 3.8: The overhead of runtime failure mitigation.

overhead in percentage. The results show a runtime performance overhead of only 1.45% on average.

Scalability. There are two aspects as far as scalability is concerned. First, our method can scale up to applications with large code bases, as shown by the LoC (Lines of Code) numbers in Table 3.1. Second, our method scales well when we increase the number of concurrent threads. In particular, we have evaluated the scalability of our method by gradually increasing the number threads and checking for the runtime overhead. Figure 3.9 shows our results on the `Boost:file_write` example, where the x-axis shows the number of concurrently running threads, and the y-axis shows the runtime overhead in percentage. Notice that the x-axis is in the logarithmic scale, whereas the y-axis is in the linear scale. Therefore, the runtime overhead of our method increases only slightly when more threads are added to the system.

Compared to the existing methods for runtime mitigation of concurrency failures, our new method has remarkably low performance overhead (on average 1.45%). To put it into context, in the AVIO-S [83] tool, their purely software based solution imposes an average slowdown of 25X. Although their AVIO-H has an overhead of 0.4-0.5%, it requires the support of a specialized microprocessor hardware. Our method does not have such restriction.

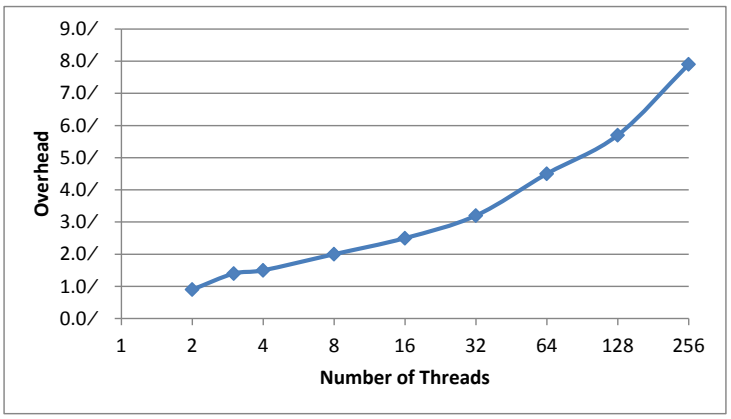


Figure 3.9: Runtime overhead on `Boost:file_write`: while the x-axis is in logarithmic scale, the y-axis is in linear scale.

The main reason why we can maintain very low runtime overhead is because we have attacked the problem at the proper abstraction level, i.e., by focusing on the method calls, rather than individual load/store instructions on global memory. In addition, we have made the following optimizations in our implementation. Our LLVM-based code instrumentation, which adds monitoring and control routines for method calls, uses a whitelist to filter out the APIs that are not defined as transitions in the type-state automaton. For example, if we target the prevention of `socket` related type-state violations, we will only add monitoring and control routines for calls defined as transitions in the automaton of `socket`, while ignoring other method calls. Because of this, the runtime overhead of our method is significantly smaller than existing methods for runtime prevention of low-level concurrency bugs.

3.8 Related Work

Type-state automaton was first introduced by Strom and Yemini [125]. Since then, many static (e.g., [23, 24, 41]) and dynamic (e.g., [10, 13]) methods have been proposed for detecting standard type-state errors. However, they do not focus on concurrency related type-state violations that are specific only to concurrent applications. To the best of our knowledge,

the only existing methods for runtime detection of concurrency related type-state violations are PRETEX [60] and 2ndStrike [44]. PRETEX can detect not only type-state violations exposed by the current execution, but also violations in some alternative interleaving of the events of that execution. However, PRETEX may have false positives. 2ndStrike improves over PRETEX by using guided re-execution to filter out the bogus violations. However, these two methods only detect concurrency related type-state violations but do not prevent them.

There exists a large body of work on mitigating low-level concurrency bugs, such as data-races and atomicity violations over shared memory reads and writes. For example, Yu and Narayansamy [158] compute certain event ordering constraints from good profiling runs and use them in production runs to restrict the thread interleaving. Their method can help avoid the previously untested (and potentially buggy) interleavings. Similar methods have also been proposed for automated healing of previously encountered atomicity violations [85, 77, 145, 58] and data-races [104, 107]. Wu et al. [150] use a set of user-specified order constraints to hot-patch a running concurrent application, by selectively inserting lock statements to the code. Dimmunix [61] and Gadara [141, 142] are tools for avoiding previously encountered deadlocks. Recent works by Liu et al. [78, 143] have extended the idea to avoid atomicity violations. Khoshnood et al. [67] develop a model checking based tool for repairing assertion failures in multithreaded programs. Although these methods have exploited the idea of restricting thread interleavings to prevent concurrency bugs, they do not focus on type-state violations. Our method, in contrast, relies on the automaton specification as correctness criterion.

Another closely related method is EnforceMop [86], which can mitigate the violation of temporal logic properties and properties such as *mutual exclusivity* in Java programs, by blocking the threads whose immediate next action violates these properties. However, EnforceMop may introduce artificial deadlocks (which have to be reported to the developers for manual debugging), and does not guarantee to find the correct interleaving even if such an interleaving exists. For the example in Figure 3.1, EnforceMop would have allowed `destroy()` to

execute right after `init()`, without foreseeing the violation caused by future events `start()` and `stop()` in the first thread. The reason is that EnforceMop only controls the immediate next actions of the threads, whereas our method can look at future events many steps ahead by analyzing the interaction of the control flow of the program and the automaton.

Concurrency bug detectors [92, 93, 138, 155, 136, 111, 132, 160, 73, 133, 74, 72, 162, 46, 75] focus on exposing the failures. In contrast, this work focuses on preventing concurrency failures by perturbing the thread interleaving – failures would not occur at run time even if the program is buggy.

At a higher level, our method for mitigating runtime failures is related to the line of works on program synthesis [35, 88, 102, 130, 16, 151] and controller synthesis [105, 106]. For example, Vechev et al. [130] infer high-level synchronizations [71] from safety specifications. Deshmukh et al. [26] require pre- and post-conditions. Deng et al. [25] and Yavuz-Kahveci et al. [156] semi-automatically map high-level models into low-level implementations. There is also a large body of work on leveraging syntax-guided synthesis (SyGuS) techniques [8, 122] to eliminate software bugs and security vulnerabilities [91, 6, 27, 31, 32, 29, 30, 28, 33, 34]. However, these methods are all offline analysis methods for generating or patching the program code, which is different from online failure mitigation. For example, it may be acceptable to spend minutes or even hours to search for a superior program code modification, if it leads to a valid solution. However, in runtime failure mitigation, the computational overhead needs to be many orders of magnitude smaller, e.g., in microseconds.

Our new method is orthogonal to the existing failure recovery methods based on checkpoint-and-rollback. Checkpointing is expensive in general, and speculative execution has its own limitations, e.g., in handling I/O. But more importantly, these methods focus on the recovery after the failure occurs, whereas our method focuses on avoiding the failure in the first place. Therefore, it is reasonable to assume that the two types of methods can co-exist in the same system. If the failure can be prevented, then there is no need for rollback recovery. If the failure becomes unavoidable, then rollback recovery can be used as the last resort.

3.9 Discussions

We have presented the first runtime method for suppressing concurrency related type-state violations in real-world multithreaded C/C++ applications. Our main contribution is the new theoretical framework within which the correctness and performance of various mitigation strategies can be analyzed. This theoretical framework is important in that it can help ensure that our mitigation actions are always safe. It also tells us when our method can guarantee to steer the program to a failure-free interleaving. We have presented a practical algorithm, which retains the safety guarantee of the theoretical framework while optimizing for performance. Our experimental results show that the new method is both efficient and effective in suppressing concurrency related type-state violations in C/C++ applications.

Chapter 4

Classifying Race Conditions in Web Applications

Modern web applications are becoming increasingly complex due to the need to implement many more features on the client side than before, while maintaining a speedy response to the users. This often requires asynchronous programming and pervasive use of JavaScript code. Although the web browser guarantees that each JavaScript code block is executed atomically, which means there is no *data-race* in the traditional sense (unlike multithreaded Java or C++ programs), high-level *race conditions* can still occur due to deferred parsing of HTML elements, interleaved execution of event handlers, timers, remote HTTP requests, and their callback routines.

Although there exist a number of tools for detecting race conditions in web applications [163, 101, 108, 52, 95], they tend to report many warnings, of which only a small portion are real in the sense that the racing events can actually affect the program state. For example, prior study shows that *EventRacer* [108] reported hundreds of warnings from the official websites of many Fortune-500 companies; although some of race conditions are harmful, the vast majority are harmless, which means directly reporting them to developers would be counter-productive. Although pattern-based techniques have been used to filter warnings,

they are extremely unreliable for separating the harmful race conditions from the harmless ones, especially when the warnings do not match any of the known bug pattern.

In this chapter, we propose the first *replay-based* method for classifying race-condition warnings in web applications, by re-executing each pair (ev_a, ev_b) of racing events in different orders and comparing the states of the web application. That is, we first execute the web application while forcing event ev_a to occur before event ev_b, and then execute it while forcing ev_b to occur before ev_a. We say that (ev_a, ev_b) is harmful only if (1) both execution orders are feasible and (2) the two resulting program states, ps_1 and ps_2, differ significantly. We say that two program states ps_1 and ps_2 differ significantly if some important fields of their HTML DOM trees, global JavaScript variables, or environment variables are different. The intuition is that, when the execution order of ev_a and ev_b is not controlled by the program logic and yet it affects the program behavior, it is a potential bug that deserves a closer look by the developers.

The overall flow of our new method is shown in Figure 4.1, which takes the URL of a website and a set of *race-condition warnings* as input, and returns the set of harmful race conditions as output. First, it uses a source-to-source transformation tool to instrument the HTML files to add self-monitoring and self-control capabilities. Then, it analyzes the warnings reported by the race detection tool and generates configuration files for deterministically replaying the web application. Next, for each pair (ev_a, ev_b) of racing events, it executes the web application twice, once with ev_a preceding ev_b and the other time with ev_b preceding ev_a. Finally, it compares the two resulting program states ps_1 and ps_2.

Toward this end, we need to overcome two technical challenges. The first technical challenge is to develop a robust method for deterministically replaying a JavaScript-based client-side web application. This is not easy due to the myriad sources of nondeterminism in practice. For example, a racing event may occur during the parsing of HTML elements, the interleaved execution of JavaScript code, the dispatch of multiple event handlers, the firing of multiple timer events, the execution of asynchronous HTTP requests, as well as the execution of their

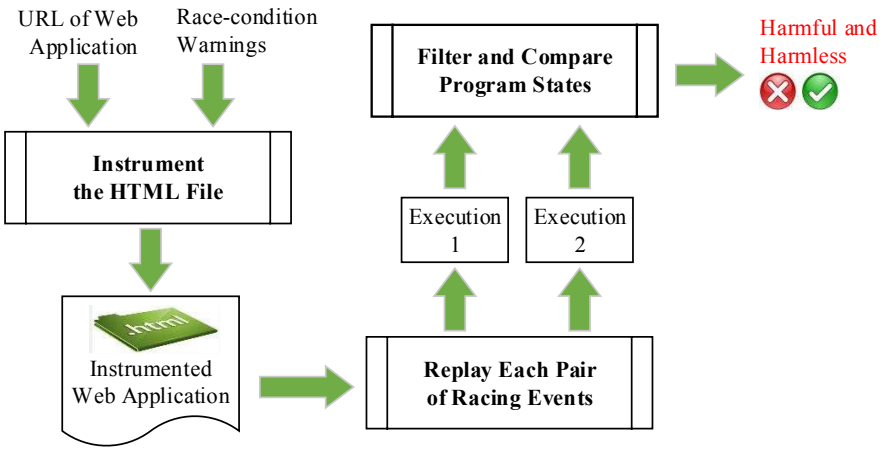


Figure 4.1: *RClassify* : The overall flow of our new method.

callback routines. Therefore, our main contribution in this context is the development of a unified framework for precisely controlling the execution order of these racing events.

An important feature of our replay framework is that it is implemented within the web application and therefore is platform-agnostic. That is, we leverage a source-to-source transformation of the HTML files to add self-monitoring and self-control capabilities to the application (see Section 4.4 for more details). This is different from most of the existing race detection tools, which rely on modifying the web browser or JavaScript execution engine. Since web browsers and JavaScript execution engines, as well as the underlying standards, are being constantly updated, any software tool implemented for a particular version of the web browser or JavaScript engine will quickly become obsolete. In contrast, since our method is platform-agnostic, it will be more robust against changes and updates of the underlying web browsers and JavaScript execution engines.

Our replay framework can decide if both of the two controlled executions are feasible. If any of them is infeasible, by definition, this is a bogus race. There can be several reasons why bogus races are reported by existing race detection tools in the first place. One reason is that some hidden happens-before relations are not accounted for by these tools since precisely capturing them is difficult in practice. Another reason is that sometimes the order of one race (ev_a, ev_b) cannot be flipped if the order of another race (ev_c, ev_d) is fixed. That is, (ev_a, ev_b)

is dependent on (ev_c, ev_d). In general, it is impossible to decide whether a *dependent* race is harmful without having to explore all possible interleavings. Furthermore, even the number of independent races is large in practice [108]. Therefore, in this work, we focus on only the *independent* races.

The second challenge is to decide, during the recording and comparison of program states ps_1 and ps_2, which fields of the program states are important and therefore should be compared. For a typical client-side web application, the number of fields in the program state can be extremely large, which means including all fields in the program state would result in a large run-time overhead. Furthermore, some fields are designed to be sensitive to sources of nondeterministic that are irrelevant to the racing events under investigation. For example, certain fields may have different values depending on the date or time of the day. In such cases, these irrelevant fields should be excluded from the state comparison; otherwise, they will cause false positives. Therefore, our main contribution in this context is the development of a flexible configuration interface allowing the user to specify which fields should be excluded. We also propose a testing-based training method for automatically identifying and excluding these irrelevant fields.

We have implemented our new method in a software tool called *RClassify* and evaluated it on a large set of benchmarks including test cases from prior publications as well as real websites from Fortune-500 companies. Our experimental results show that *RClassify* outperforms existing techniques such as *EventRacer* [108] and a recent tool developed by Mutlu et al. [95]. In particular, *RClassify* can robustly identify all 33 known-to-be-harmful races from the 50 warnings in the test cases, whereas the tool by of Mutlu et al. [95] cannot identify any of them. Furthermore, *RClassify* can efficiently handle the large number of warnings from real websites. For the 70 real websites of Fortune-500 companies that we tested, *EventRacer* [108] returned 1,903 warnings, among which our *RClassify* identified 73 as bogus, 132 as harmful, 1644 as harmless, and 54 as undecided. We have manually reviewed these results and confirmed the correctness of our classifications.

To sum up, this chapter makes the following contributions:

- We propose the first *replay-based* method for classifying race-condition *warnings* in web applications, by re-executing the racing events in different orders and observing their actual impact.
- We develop a purely JavaScript-based framework for robustly replaying racing events of web applications. Since it is platform agnostic, it can be used for a wide range of other applications including testing and debugging.
- We conduct experiments on a large number of benchmarks to demonstrate the effectiveness and efficiency of our method in handling real-world applications.

4.1 Motivating Examples

In this section, we use examples to illustrate the main ideas behind our method while highlighting the technical challenges.

4.1.1 Race Conditions

Consider the web page in Figure 4.2, which contains an image, a button, a JavaScript code block, and other HTML elements omitted for simplicity. The *image.onload* event, fired after the browser downloads *image1*, invokes the *image1Loaded()* function, which in turn attaches a listener function to the *onclick* event of *button1*. The button may be clicked by the user immediately after it is parsed. The listener function, named *func()*, changes the text in *outputField* to `'Well done!'` Therefore, the expected event sequence is ev_0*: parsing(image1)* $\rightarrow$ ev_1*: parsing(script1)* $\rightarrow$ ev_2*: parsing(button1)* $\rightarrow$ ev_3*: firing(image1.onload)* $\rightarrow$ ev_4*: parsing (outputField)* $\rightarrow$ ev_5*: firing(button1.onclick).*

However, depending on the network speed, web browser's parsing speed, and timing of the user click, there may be more than one alternative execution sequences, some of which do

```
<html>
  <head> </head>
  <body>
    <img src="image1.jpg" onload="image1Loaded()" id="image1">
    <-- ... omitted elements ... -->
    <script id="script1">
      function image1Loaded() {
         document.getElementById("button1")
         .addEventListener("click", func);
      }
      function func() {
         document.getElementById("outputField")
         .innerHTML = "Well done!";
      }
    </script>
    <-- ... omitted elements ... -->
    <button id="button1">button1</button>
    <-- ... omitted elements ... -->
    <div id="outputField"></div>
  </body>
</html>
```

Figure 4.2: An example web page with multiple race conditions.

not lead to the expected display of `'Well done!'` As shown in the partial order of events in Figure 4.3, there are four possible race conditions:

1. **RC1** is (ev_1, ev_3) over *image1Loaded*
   (a) event ev_1: parsing(script1)
   (b) event ev_3: firing(image1.onload)
2. **RC2** is (ev_2, ev_3) over *button1*
   (a) event ev_2: parsing(button1)
   (b) event ev_3: firing(image1.onload)
3. **RC3** is (ev_3, ev_5) over *button1*
   (a) event ev_3: firing(image1.onload)
   (b) event ev_5: firing(button1.onclick)

4. **RC4** is (ev_4, ev_5) over *outputField*

   (a) event ev_4: parsing(outputField)

   (b) event ev_5: firing(button1.onclick)

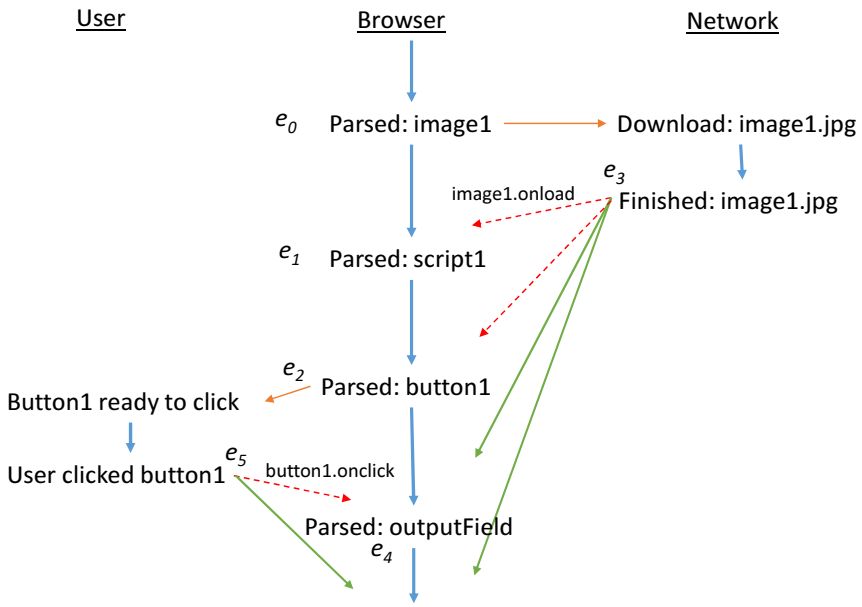


Figure 4.3: The partial order of the racing events in Figure 4.2.

The first race condition (RC1) is between the parsing of HTML element *script1* and the firing of *image1.onload* event. Typically, the parsing event finishes first, but if *image1* is downloaded before the parsing finishes, e.g., due to slow parsing of other HTML elements preceding *script1*, *image1Loaded* will be undefined when the browser invokes it through *image1.onload*.

The second race condition (RC2) is between the parsing of *button1* and the firing of *image1.onload*. Here, we assume the unwanted situation regarding RC1 did not occur (ev_1 precedes ev_3 and *imageLoaded* is properly defined). In this case, it is possible for *image1.onload* to happen before the parsing of *button1*, thereby causing *document.getElementById ("button1")* at Line 8 to fail.

The third race condition (RC3) is between the firing of the *image1.onload* event and the

firing of *button1.onclick*. Here, we again assume the unwanted situations regarding RC1 and RC2 did not occur (both ev_1 and ev_2 precede ev_3). However, it is possible for *button1* to be clicked before the firing of *image1.onload*. Since *func()* has not yet been attached to *button1.onclick*, clicking the button will not lead to the desired behavior.

The fourth race condition (RC4) is between the parsing of HTML element *outputField* and the firing of *button1.onclick*. Here, we again assume that none of the unwanted situations regarding RC1, RC2, and RC3 occurred. However, it is still possible for *button1* to be clicked and yet the `'Well done!'` message is not displayed. This occurs when the button is clicked before the parsing of *outputField*, causing *document.getElementById ("outputField")* at Line 12 to fail.

The example in Figure 4.2 shows that even a simple web page can have many race conditions. The situation will become significantly worse in practice when the race detection tools report too many warnings, since it would be counter-productive if we report them directly to developers: They would have to spend long hours weeding out the bogus warnings. In this work, our goal is to lift the burden and allow developers to focus on diagnosing the truly harmful race conditions.

4.1.2 Harmful Race Conditions

We leverage deterministic replay, together with state recording and state comparison, to classify race-condition warnings into harmful and harmless ones. For each pair (ev_a, ev_b) of racing events reported by the race detectors, we perform controlled executions of the web application twice. In the first execution, we make sure that ev_a precedes ev_b. In the second execution, we make sure that ev_a precedes ev_b. If any of these executions is infeasible, the replay will fail, indicating that it is a bogus warning. Otherwise, we record the program states at the end of the two executions and compare them. If the program states differ significantly, we classify (ev_a, ev_b) as a harmful race condition.

Consider RC1 in Figure 4.2 as an example. First, we run the application while forcing ev_1 to occur before ev_3, which leads to the expected sequence. To decide whether RC1 is harmful, we run the application again while forcing ev_3 to occur before ev_1. We observe that both executions are feasible, and the second execution invokes *image1Loaded* before it is defined, thereby causing the web browser to print an error message in the console window. The event handler for *button1.onclick* also becomes uninitialized, subsequently causing a difference in *outputField*. Due to these differences in the program states, we classify RC1 as a *harmful* race, meaning that it deserves the attention of the developers.

However, not all race conditions are harmful. Consider the three race conditions in Figure 4.4, which we detected from the official website of *www.newyorklife.com*. The race between parsing of *utag.js* and firing of *document.onDOMContentLoaded* is harmless, but the race conditions over *input_1* and *input_2* are harmful. The first race is harmless because, although the event handler of *document.onDOMContentLoaded* may be registered (by *utag.js*) after the browser fires the *document.onDOMContentLoaded* event, the same callback function *c()* is also registered to *window.onload* as a precaution. Since *window.onload* always execute after the load of *utag.js*, it ensures that *c()* is always executed, thereby resulting in the same program state.

However, the two races over *input_1* and *input_2* are harmful, because *input_1* and *input_2* initially show the hint values `'Name'` and `'Email'` to the user, but their *onfocus* event handlers, which call *clearText()*, will empty these hint values as soon as the user tries to type into the text areas. Unfortunately, the script that defines *clearText()* may be parsed after the user typed something into the two fields – assume `'USERTYPED'` is what the user typed. This can lead to unwanted text contents such as `'NamUSERTYPEDe'` or `'NameUSERTYPED'` instead of just `'USERTYPED'`. Our new method can correctly classify these two race conditions as harmful because there are significant differences in the resulting program states. In contrast, *EventRacer* [108] would report all three races as warnings to the user, whereas the method of Mutlu et al. [95] would not be able to detect any of them.

```
<-- www.newyorklife.com -->
...
<input id="input_1" type="text" value="Name" onfocus="clearText(this)">
<input id="input_2" type="text" value="Email" onfocus="clearText(this)">
<script id="script2" src="nyl-min.js"></script>
<script id="script1">
  //insert async script to the DOM tree, the src = "utag.js"
  ...
</script>
...
<-- nyl-min.js -->
...
function clearText(a){if(a.defaultValue==a.value){a.value=""}}
...
<-- utag.js -->
  function c(){
   if(!done){
   done = true;
   //execute some code
  }
}
document.addEventListener("DOMContentLoaded", c, false);
window.addEventListener("load", c, false);
```

Figure 4.4: Race conditions detected in www.newyorklife.com.

4.2 Preliminaries

In this section, we provide the background of client-side web applications and the definition of race conditions.

4.2.1 Web Applications

A modern HTML page consists of a set of elements, each of which has an opening tag and a closing tag, together with the content in between, e.g., `<p>...</p>` for a paragraph element. Elements may be declared statically in the HTML file or inserted dynamically by executing JavaScript code. The document object model (DOM) is a tree representation of the web page to be rendered by the web browser. Each node in the DOM has attributes for

holding meta-data, e.g., the `<img>` element has an `src` attribute indicating the URL of the image. JavaScript code may be embedded in the HTML file, or declared externally via the `src` attribute, e.g., `<script src="code.js"></script>`. By default, they are synchronous scripts in that the browser must wait until each script is downloaded and parsed before moving to the next HTML element. However, external scripts may be declared using the "async" or "defer" attribute. A deferred script will run after all the static HTML elements are parsed, while an asynchronous script may run at any time after it is downloaded.

Client-side web applications are written in an event-driven programming style, where various event handlers are registered to the DOM nodes to react to user interactions as well as changes to the corresponding HTML elements, such as *element.onload*. An event may be be propagated through the DOM tree through "capturing" and "bubbling". For example, based on the W2C standard specification, if a button is clicked, the *onclick* event will be fired not only for this button, but also for all other DOM nodes that recursively contain this button. The propagation is performed level by level, all the way up to the *window* object. If there is any *onclick* event handler registered to any of these DOM nodes along this chain, it will also be fired.

The execution of a web application can be represented by a sequence of events, denoted $\rho = ev_1, \ldots, ev_n$. Each event is of the form $ev = \langle id, type, info, mem \rangle$, where id is the unique identifier of the event, $type$ is the event type, $info$ stores additional information of the associated DOM event, and mem stores information of the shared memory accesses. Each event can have one of the following types:

- $parse(element)$, which represents the parsing of non-script element. If the elements are static, they will be parsed sequentially (one after another) following the order in which they are declared in the HTML file. If the element is an embedded HTML file, the parsing may be interleaved with other parsing or JavaScript execution events.
- $execute(js)$, which represents the parsing and execution of an embedded JavaScript code block or the execution of a deferred or asynchronous script.

- $fire(eh\_cb)$, which represents the execution of a callback function eh_cb dispatched in response of an event, such as the *onclick* event of a button.
- $fire(tm\_cb)$, which represents the execution of a callback function tm_cb dispatched in response of a timer, registered using either *setInterval()* or *setTimeOut()*.
- $fire(ajax\_cb)$, which represents the execution of a callback function $ajax\_cb$ dispatched in response to an AJAX request.

The program state $ps = \langle DOM, JS, Env, Console \rangle$ is a snapshot of the run-time memory of the web application, where DOM represents the DOM tree, JS represents values of JavaScript variables, Env represents the values of environment variables, and $Console$ represents the console messages. Let PS be the set of all possible program states. The execution trace of a web application is represented by $\lambda = ps_0 \xrightarrow{ev_1} ps_1 \xrightarrow{ev_2} ps_2 ... \xrightarrow{ev_n} ps_n$, where $ps_1, \ldots, ps_n \in PS$ are program states, $ev_1, \ldots, ev_n$ are events, and for each $1 \leq i < n$, we have $ps_i \xrightarrow{ev_i} ps_{i+1}$. That is, executing ev_i moves the application from ps_i to ps_{i+1}.

4.2.2 Race Conditions

Since the web browser ensures that each event of the web application is executed atomically, it is impossible for individual statements of one JavaScript code block to interleave with statements of another JavaScript code block. In this sense, web applications doe not have *data-races* in the traditional sense (unlike multithreaded Java or C++ programs). However, they can still have *race conditions*. For example, a web application may consider fast rendering of some visual elements as a priority, while asynchronously loading and processing some images, audio, video, and JavaScript code libraries in the background. It is possible that event handlers of some of these visual elements are attached after the elements are made available to the user, while their associated JavaScript functions have not yet been defined.

Before defining the race condition formally, we define the must-happens-before relation, denoted $\rightarrow_{mhb}$. It is a binary relation over events, such that $(ev_a, ev_b) \in \rightarrow_{mhb}$ if and only if ev_a

precedes ev_b in all possible executions of the web application. In contrast, if ev_a precedes ev_b only in one execution, we say $ev_a \prec ev_b$. Given an event ev, we use $WR_{var}(ev)$ to denote the set of memory locations written by ev, and $RD_{var}(ev)$ to denote the set of memory locations written by ev. A race condition is defined as a pair (ev_a, ev_b) of events such that

1. $(ev_a, ev_b) \not\in \rightarrow_{mhb}$, $(ev_b, ev_a) \not\in \rightarrow_{mhb}$, and
2. $\exists var$ such that $var \in WR(ev_a) \cap WR(ev_b)$ or $var \in WR(ev_a) \cap RD(ev_b)$ or $var \in RD(ev_a) \cap WR(ev_b)$.

However, a race condition (ev_a, ev_b) may not be considered as harmful if it does not make a difference in the result [7]. More formally, we say that a race condition is harmful only if there exist two concrete execution traces λ_1 and λ_2 such that (1) $ev_a \prec ev_b$ in λ_1, (2) $ev_b \prec ev_a$ in λ_2, and (3) the two program states ps_1 and ps_2 (resulting from λ_1 and λ_2 respectively) are different.

Although there already exist a number of race detection tools for web applications [101, 108, 52, 94, 95], they either produce too many warnings or miss many truly harmful race conditions. For example, *EventRacer* [108] uses an instrumented version of WebKit to record the execution trace of an application and then analyzes the trace offline using a happens-before causality model. WAVE [52] detects race conditions by creating a sequence of external events and feeding it to the web application; it then checks whether a different test case (sequence of external events) can produce a different result. More recently, Mutlu et al. [95, 94] proposed a method for detecting only harmful races using a combination of dynamic analysis and trace-based static analysis. They first leverage dynamic analysis to obtain an execution trace and then apply predictive static analysis to explore the alternative interleavings in order to estimate the impact of the race. However, none of these existing methods is based on replay and program-state comparison. As such, they do not have the same level of precision as our new method in identifying the truly harmful race conditions.

4.3 Algorithm

Our method takes the URL of the web application and a set of race-condition warnings as input and returns the classification results as output, as shown in Figure 4.1. It reports only the race conditions for which both execution orders of the racing events are feasible (or real). After removing the bogus races, it divides the remaining (real) races into two categories: the harmful ones and the harmless ones. Harmful races are those that affect the program states, whereas harmless races are those that do not affect the program states.

Specifically, our method first downloads the web application following the given URL and instruments the HTML files using a source-to-source transformation to add self-logging capabilities. The goal is to record information of the racing events so that they can be identified and controlled during the deterministic replay. During the experiments, we use the existing race detection tool *EventRacer* to generate the race-condition warnings (our input); however, other similar tools may also be used. Next, our method classifies these warnings using replay and program-state comparison. To classify the warnings into the three groups (bogus, harmful, and harmless), it executes the application twice for each pair (ev_a, ev_b) of racing events. In the first execution λ_1, it forces ev_a to occur before ev_b. In the second execution λ_2, it forces ev_b to occur before ev_a. During both executions, we maintain the execution order of all other racing events as much as possible. If both executions are feasible, we conclude that (ev_a, ev_b) is a real race condition. In such case, we record the program states at the end of both executions. Let ps_1 and ps_2 be the two program states resulting from λ_1 and λ_2, respectively. We say that the race condition (ev_a, ev_b) is harmful if there are significant differences in ps_1 and ps_2.

In the remainder of this section, we explain how to instrument the HTML file to add self-logging capabilities and how to analyze the race warnings to prepare for the replay. In the next two sections (Sections 4.4 and 4.5), we will present our techniques for controlling the execution order of racing events during replay and comparing the program states.

4.3.1 Instrumenting the HTML File

One input of our method is the URL of the web application, consisting of the HTML source file together with all embedded resource files, such as Cascading Style Sheets (CSS), JavaScript code, image, audio, and video. After fetching these files, we instrument the HTML before using it as input of *EventRacer* for race detection. We use an open-source HTML parser called *Jsoup* (http://jsoup.org/) to insert our own library *Pre_RClassify.js*, to the HTML *head* element. This JavaScript code block will be executed before the browser loads the content of the web page, thereby allowing us to track the execution of all racing events at run time.

Our instrumentation of the HTML file is designed to collect information about the race-condition warnings so that they can be replayed deterministically. Figure 4.5 shows an example of the instrumented web page, where Lines 3 and 10-12 are added. They execute *Pre_RClassify.js* before loading the actual web page. During the execution of *Pre_RClassify.js*, we redefine APIs functions to intercept calls to *addEventListener()* and *setTimeOut()*. At run time, in addition to invoking the original handlers, the instrumented code will also gather information of these racing events for later phases of our algorithm. In addition, we also insert a unique *id* attribute to each HTML element if it does not already have one. This *id* attribute will help pinpoint the HTML element involved in the racing event and therefore allow us to control its execution order during replay.

During the instrumentation, we also insert a *window.onload* event handler at the end of the HTML *body* element, which is fired by the web browser after the page is fully loaded. The handler function *RC_fire_event_handlers()* goes through all active event handlers that require user actions and fires them one by one, to simulate user interactions. By automatically dispatching these active event handlers, as opposed to relying on the user to click buttons and provide text inputs, we can improve the coverage of existing race detection tools. For example, *EventRacer* can only dispatch a limited number of event handler types, whereas our new method can dispatch any kind of handler as specified in the W3C DOM Level 2

```
<html>
  <head>
    <script src="Pre_RClassify.js"></script>
    <!-- here begins the elements -->
    <-- ... -->
 </head>
 <body>
   <-- ... -->
    <-- after all the elements -->
    <script>
     window.addEventListener("load", RC_fire_event_handlers, false);
    </script>
  </body>
</html>
```

Figure 4.5: Instrumented web page prior to the race detection.

standard.

4.3.2 Analyzing Race Condition Warnings

After using *EventRacer* to generate the set of warnings, which serve as input to *RClassify* , we need to analyze them and prepare for the replay. *EventRacer* uses an instrumented version of WebKit to generate a trace log while loading the web page. Then, it analyzes the trace log offline to build a happens-before causality model. The model captures all the shared resources, i.e., DOM elements and JavaScript variables, that have conflicting memory accesses from concurrent events. Two accesses to the same memory location are conflicting if at least one of them is a write operation. However, *EventRacer* cannot robustly separate harmful races from harmless ones. For example, it reported hundreds of warnings on the official websites of many Fortune-500 companies [108], most of which are bogus or harmless races.

Our focus is on applying the replay-based approach to automatically classifying race-condition warnings. For each $RC(ev_a, ev_b)$, we first check if both execution orders of the racing events are feasible. Toward this end, we use *JSoup* to generate two instrumented versions

of the HTML file and add the self-control capability. In one version, we make sure that $ev_a \prec ev_b$. In the other version, we make sure that $ev_b \prec ev_a$. To prevent interference from other pairs of racing events, we force these other racing events to maintain their original execution order instead of allowing them to interleave freely. For example, given two race-condition warnings $RC_1(ev_a, ev_b)$ and $RC_2(ev_c, ev_a)$, and the first execution trace $\lambda_1 = ps_0 \ldots ps_i \xrightarrow{ev_c} \ldots ps_j \xrightarrow{ev_a} \ldots ps_k \xrightarrow{ev_b} \ldots ps_n$ where $ev_a \prec ev_b$. To classify RC_1, we would obtain the second execution trace $\lambda_2 = ps_0 \ldots ps_i \xrightarrow{ev_c} \ldots ps_j \xrightarrow{ev_b} \ldots ps'_k \xrightarrow{ev_a} \ldots ps'_n$, where $ev_b \prec ev_a$. In both execution traces, we make sure that the order $ev_c \prec ev_a$ is maintained for RC_2.

Figure 4.6 shows an example of the instrumented HTML file. Similar to Figure 4.5, we load another library, *RClassify.js*, and information of the racing events at the beginning of the *head* element. The JavaScript file contains functions to control the execution order of racing events and record the program state. In the HTML *body* element, we also insert a JavaScript code snippet for invoking the function *RC_detect_handler_changes()* after the parsing of every HTML element (Line 11). This function checks whether the list of event handlers attached to any element of the DOM tree has changed. If new event handlers are added, we retrieve and instrument them so that we can track their executions at run time.

When loading the order of racing events for replay (Line 3), if the event ev_a needs to be executed before ev_b, we put ev_a into the $toWaitList[ev_b]$. During replay, we monitor all racing events dynamically and force ev_b to wait for all events in $toWaitList[ev_b]$ to complete before executing ev_b. When there are multiple race-condition warnings, however, it may not always be possible to maintain the execution order of all other racing events while flipping the race condition under investigation. The reason is that reversing the order of this race condition may invalidate the order of the other race conditions.

For example, consider the races $RC_1(ev_a, ev_c)$, $RC_2(ev_a, ev_b)$ and $RC_3(ev_b, ev_c)$ in an execution where $ev_a \prec ev_b \prec ev_c$. When reversing RC_1, it is impossible for us to maintain the original order of both RC_2 and RC_3. In such cases, we try to maintain the order of as many

```
1  <html>
2    <head>
3      <script> //the replay info of all races </script>
4      <script src="RClassify.js"></script>
5      <!-- here begins the elements -->
6      <-- ... -->
7    </head>
8    <body>
9      <-- ... -->
10     <element m>
11     <script>  RC_detect_handler_changes();</script>
12     <-- ... -->
13     <-- after all the elements -->
14     <script> window.addEventListener("load", RC_dump_state, false);
15     </script>
16   </body>
17 </html>
```

Figure 4.6: Instrumented web page for deterministic replay.

of the other race conditions as possible. That is, after loading the ordering information of the reversed RC_1 and the original RC_2 into the HTML *head* element, we would discover that they conflict with the original order of RC_3. Therefore, we ignore RC_3 and obtain an execution where $ev_c \prec ev_a \prec ev_b$.

When the must-happens-before relation among race conditions are available, we use it to further refine the execution order information needed for replay. For example, if there is a must-happens-before relation $ev_a \rightarrow_{mhb} ev_b$, we know that the order of ev_a and ev_b cannot be flipped. Although analyzing RC_1 or RC_2 alone would not detect any conflict, when reversing RC_1, we know that it is no longer possible to maintain both RC_2 and RC_3. Therefore, we ignore RC_2 and obtain an execution where $ev_c \prec ev_a \prec ev_b$.

During instrumentation, we also insert a *window.onload* handler at the end of the HTML *body* element (Lines 14-15). At the end of each execution, the handler function *RC_dump_state* saves all relevant fields of the program state into a disk file. After replaying the racing events and recording the two program states, we compare them to decide whether the race is harmful. We say that a race condition (ev_a, ev_b) is harmful if the two program states ps_n and

ps'_n are significantly different. Toward this end, an important problem is to identify and filter out the fields of the program states that may be affected by nondeterminism sources other than the racing events, since failure to do so will lead to harmless races to be incorrectly classified as harmful.

In the next sections, we explain in detail how to accurately control the order of racing events during replay and how to compare relevant fields of the program states.

4.4 Controlled Execution

Deterministically replaying the racing events of a web application is a main challenge, and being able to intercept these racing events is a prerequisite for controlling their execution order. In this section, we explain how to intercept the callback functions of these racing events and how to control their execution order.

4.4.1 Intercepting the Callback Functions

We need to intercept the registration, modification, and removal of callback functions for handling events, timers, as well as asynchronous AJAX requests. For each callback function, we replace it with a wrapper function that, prior to invoking the original callback function, checks whether it is the right time to execute the callback or whether the execution should be postponed.

Broadly speaking, there are two types of callback functions for event handling in client-side web applications. Type 1 handlers are installed by setting a DOM element's attribute such as *el.onload* and *el.onclick*. Type 2 handlers, in contrast, are added and removed by calling *addEventListener()* and *removeEventListener()*, respectively. Each element may have more than one Type 2 handlers; they are store inside the browser as opposed to the DOM tree itself. Therefore, they cannot be accessed by traversing the DOM tree. Furthermore,

Algorithm 6 Scanning and instrumenting new event handlers.

```
1: RC_detect_handler_changes () {
2:     for each (el in document.all_elements) {
3:         for each (eh_type in all_event_handler_types) {
4:             if (el[eh_type] has changed) {
5:                 event_str = compose_es(el.id, eh_type);
6:                 orig_func = el[eh_type];
7:                 if (is_racing_event(event_str)) {
8:                     el[eh_type]= replace_callback(event_str, orig_func);
9:                 }
10:            }
11:        }
12:    }
13: }
```

the order in which Type 2 handlers are fired is the same as the order in which they are added. Third-party libraries such as jQuery often define their own APIs for accessing event handlers, such as *jQuery.bind()* and *jQuery.detach()*. But underneath, they still rely on the two aforementioned methods.

When Type 1 event handlers are added and removed through the use of attributes, such as *el.onload* and *el.onclick*, they are difficult to intercept at run time. The reason is that we are not modifying the underlying web browser or intercepting the execution of each individual statement of the JavaScript code block. Instead, our instrumentation is at the HTML file level. Since these handlers may be added during the parsing of any HTML element and modified by the execution of any JavaScript code that accesses *el.onclick*, it is difficult to know when handlers are added or removed. However, we need to know it in order to precisely control the execution order.

To solve this problem, we developed a unified framework for detecting changes to event handlers, which periodically scans the DOM tree and compares it with the DOM tree recorded during the previous scan. If there is any change in any DOM element's Type 1 event handler, e.g., the addition, deletion, or modification of *el.onclick*, we will be able to detect it. In Figure 4.6, this is implemented inside *RC_detect_handler_changes()*. By instrumenting the HTML file, we can invoke this function after the parsing of every static HTML element, and after invoking callback functions that may modify the DOM tree.

The pseudocode of *RC_detect_handler_changes()* is shown in Algorithm 6. After detecting a change of *el[eh_type]*, which corresponds to the event *el.eh_type*, we instrument the handler as follows: *var orig_func = el.onclick; el.onclick = replace_callback (event_str, orig_func)* { *generate and return RC_func; ...*}, where *RC_func* is a dynamically generated instance for *orig_func*, in addition to monitor and control the execution order, *RC_func* also invokes the original callback function *orig_func* eventually. The *replace_callback()* function can be seen as a factory function. It creates an instance of the wrapper function *RC_func* for the given *event_str* and $orig\_func$, and replaces the callback function with the wrapper *RC_func*.

Compared to Type 1 event handlers, Type 2 event handlers are easier to intercept. The list of active Type 2 handlers can be constructed by intercepting *addEventListener()* and *removeEventListener()*. Recall that while executing *RClassify.js*, we have replaced them with our own versions, e.g., by executing *var orig_addEL = addEventListener; addEventListener = function new_addEL()* {*orig_addEL();...*} At runtime, instead of invoking the original API function, executing the JavaScript code would invoke *new_addEL()*. It does not add *orig_func* as an event handler to *element*; instead, it first dynamically creates an instance of the wrapper function *RC_func* and then adds it as the event handler callback function to *element*.

4.4.2 Controlling the Execution Order

Once the callback function *orig_func()* is replaced by the wrapper function *RC_func()*, its execution can be selectively postponed such that its relative order with other events can be enforced. The wrapper *RC_func* is instrumented according to Algorithm 7. Each of the callback function *orig_func()* is replaced by a dynamically generated instance of *RC_func()*. If not all events in the racing event's *toWaitList* are already executed, means the racing event should be postponed and its *RC_func* will be put into the *postponedEvent* list, which will be checked periodically and executed when the right time comes. In other words, we control the event execution order of callback function by conditionally postponing certain

events. When a racing event is about to be dispatched, instead of executing the original function directly, we will call the function *RC_func*, which makes sure *orig_func* is ready to execute. If the event needs to be postponed, the function *RC_func* will be put into the waiting list again, until it no longer needs to wait.

Algorithm 7 Executing callback functions in the given order.

```
 1: replace_callback (event_str, orig_func) {
 2:     function RC_func() {
 3:         if (orig_func is defined) {
 4:             if (¬racing_event_finished_waiting(event_str))
 5:                 postponedEvent.push(RC_func);
 6:             else
 7:                 orig_func.call(arguments);
 8:         }
 9:     }
10:     return RC_func;
11: }
```

To ensure that replay does not deadlock, we use a counter inside each *RC_func* (not shown in Algorithm 7 for brevity) to record the number of times it has been postponed. When the counter reaches a pre-determined threshold, say 200, we assume that it has waited too long and declare that replay failed. In such case, it will stop waiting and enter a free run. The reason why replay may fail is because some racing events cannot be flipped (e.g., when the race is actually bogus). In this sense, our method has the capability of detecting bogus races. However, this may not be an efficient way to detect bogus races, since waiting for replay to fail consumes a significant amount of time.

Therefore, we also developed a cheaper mechanism to identify races that are obviously bogus, so we do not even need to replay them. Recall that bogus races are largely due to limitations of the race detector in modeling certain must-happen-before relationships. For example, there are several types of event handlers such as *document.onDOMContentLoaded* and *window.onload*, whose execution time is somewhat fixed. Specifically, the first one is fired only after the entire HTML is parsed, but without waiting for the asynchronous resources; whereas *window.onload* is fired only after the HTML is parsed and all resources are loaded. For example, if a reported race is between the parsing of an HTML element and

document.onDOMContentLoaded or *window.onload*, then it is bogus and we do not need to replay it because the order of the two racing events are fixed and therefore cannot form a race.

4.5 Program State Comparison

We record the program states after both executions and then compare their fields.

4.5.1 State Recording

After the web page is fully loaded, we serialize the program state and store the result in a disk file. We record the program state when the following two conditions are met: (1) The web page is loaded, and (2) all racing events have finished executing. We consider the following fields as parts of the program states:

1. *The DOM*: As part of the DOM, we record all HTML elements, their attributes, and the Type 1 event handlers. We also intercept all Type 2 event handlers and treat them as part of a special attribute in the DOM.
2. *JavaScript variables*: We record values of all JavaScript variables since they may affect the application behaviors.
3. *Environment variables*: We record the environment variables associated with the web browser, such as the height and width of the window.
4. *Console messages*: We record runtime information displayed in *console.log*, *console.warn* and *console.error*. They are invisible to users but useful to developers.

All functions and variables from our JavaScript instrumentation library are defined under a specific name space similar to jQuery, and therefore can be excluded from the program state easily.

To transform the above data fields into a string that can be easily compared, we use the *JSON.stringify()* API. However, JSON does not directly handle data with cyclic dependency, which are common in web applications. To solve this problem, we implemented a JavaScript method that traverses an object in the BFS order and marks each visited node with a unique identifier. If we encounter a visited object again, we replace the reference to that object with its unique identifier. Therefore, the resulting representation is guaranteed to by acyclic and can be converted to a string using *JSON.stringify()*.

4.5.2 State Comparison

To compare the two program states, we first use *JSON.parse* to restore them from the disk files and construct two key-value tables. For the DOM, we use the element's *id* attribute as the key and the HTML content as the value. For JavaScript and environment variables, we use the variable name as the key. For console messages, we use their order as the key and the logged content as the value. We say that the race condition is *harmless* if the two states are identical. We say that the race condition is *harmful* if they differ in the DOM or JavaScript variables. If they differ only in console logs or environment variables, we assume the race condition is likely *harmless* but we still report it as a warning.

Sometimes, certain fields of the DOM or JavaScript variables may have nondeterministic values due to reasons other than race conditions. For example, a web application may record the last time it is executed or create a session identifier that is different every time it is executed. To exclude such fields in state comparison – since they may lead to false positives – we developed a mechanism for uses to specify which fields should be excluded. We also developed a heuristic method for filtering out such irrelevant fields. Our solution is to execute the web application three times, with the following event orders: $ev_a \prec ev_b$, $ev_a \prec ev_b$, and $ev_b \prec ev_a$. After that, we use a three-way comparison to check the state differences. If there exists a field that has different values in all three states, we consider it as an irrelevant field. If a field is the same in the first two states, but different in the third state, we consider it as

a relevant field.

4.6 Experiments

We have implemented the replay-based method for classifying race-condition warnings in our software tool *RClassify*. It builds upon a number of open-source software: Our libraries for monitoring and control racing events is written in 2.1K lines of JavaScript code. Our HTML instrumentation component is implemented using *Jsoup 1.8.1* with 2.4K lines of Java code. We leverage *EventRacer* to generate the race-condition warnings (our input). During the experiments, we store the benchmarks under test in a local web server. For real websites, we use *Teleport Ultra 1.69* to download the source files before instrumenting them. We use *Mozilla Firefox 40.0* as the web browser. All experiments were conducted on a machine with Intel Core i7-4700 2.4 GHz CPU running 64-bit Ubuntu virtual machine, with 4GB RAM.

The benchmarks used in our experiments fall into two groups. The first group consists of manually created web application examples from recent publications such as EventRacer [101, 108] and WAVE [52] (together with its benchmarks). Since these benchmarks are specifically for illustrating various types of race conditions, the vast majority of them are known to be harmful. Our goal is to confirm that *RClassify* can correctly identify these harmful races. The second group consists of real websites of 70 randomly chosen companies on the Fortune-500 list. Recall that previously *EventRacer* was applied to the same type of websites and reported a very large number of warnings. Therefore, our goal is to see if *RClassify* can do better than existing tools such as *EventRacer* [108] and Mutlu et al. [95], i.e., whether it can correctly classify these warnings while maintaining a reasonable runtime performance.

4.6.1 Results on the Small Benchmarks

Table 4.1 shows our results on the small benchmarks. Columns 1-2 show the benchmark name and number of race-condition warnings (our input). Column 3 shows the number of bogus races that we identified. Column 4 shows the number of harmful races. Columns 5-8 show the number of harmless races. Column 9 shows the number of undetermined cases (Undet.). Column 10 shows the total time taken, including the HTML instrumentation, race analysis, replay, and state comparison. Note that some of these 50 input warnings cannot be generated by *EventRacer* due to its own limitations; in such cases, we manually created the warnings before giving them to *RClassify*. We checked all 50 warnings and manually confirmed that *RClassify* produced the correct classification.

Table 4.1: Experimental results on the small benchmarks.

Name	Warning (input)	Bogus	Harmful	Harmless			Undet.	Time	[95] Harmful	[55] Harmful
				con.	b/c/t	none				
WebR_ex1	1	0	0	1	0	0	0	23.8s	0	0
WebR_ex2	1	0	1	0	0	0	0	23.8s	0	0
WebR_ex3	1	0	1	0	0	0	0	23.8s	0	0
WebR_ex4	1	0	1	0	0	0	0	23.8s	0	0
WebR_ex5	1	0	0	1	0	0	0	24.1s	0	0
EventR_ex1	1	0	0	1	0	0	0	23.8s	0	0
EventR_ex2	1	0	1	0	0	0	0	23.8s	0	0
EventR_tut	3	0	3	0	0	0	0	1m10s	0	1
kaist_ex1	1	0	0	1	0	0	0	23.8s	0	1
kaist_ex2	1	0	0	1	0	0	0	25.0s	0	(0) 1
kaist_ex3	3	0	1	2	0	0	0	1m10s	0	0
kaist_case1	6	0	4	1	1	0	0	2m20s	0	1
kaist_case2	3	0	3	0	0	0	0	47.2s	0	1
kaist_case3	4	1	2	1	0	0	0	1m35s	0	1
kaist_case4	7	4	3	0	0	0	0	1m51s	0	0
kaist_case5	3	0	3	0	0	0	0	1m11s	0	1
kaist_case6	3	0	2	1	0	0	0	24.3s	0	1
kaist_case7	7	0	6	1	0	0	0	2m50s	0	1
kaist_case8	2	0	2	0	0	0	0	47.6s	0	1
Total	**50**	5	**33**	11	1	0	0		**0**	**(9) 10**

The 13 harmless races fall into three groups: *none* means there are no differences in the program states; *con.* means there are differences only in the *console* logs; and *b/c/t* means the differences are due to event "bubbling" and "capturing" or a "timer-try" pattern. Event bubbling and capturing (b/c) lead to duplicated events on the upper-level elements. For

example, if there are 100 buttons in a window, each of these 100 *button.onclick* events can be bubbled up to trigger *document.onclick*. In such case, race detection tools may report 100 races on *document.onclick*. Timer-try (t) is when one function repeatedly postpones its execution via *setTimeout()* until some condition is satisfied. The nature of these races means they are likely harmless.

We also compared our result with the result of running the tool from Mutlu et al. [95] and Jessen et al. [55]. [95] was designed to report only harmful races. Unfortunately, it identified 0 harmful races despite that 33 of them are known to be harmful. This is perhaps because their method was designed to handle only races involving AJAX calls and races that change the persistent states (e.g., *cookie*). In contrast, our new method is capable of handling a wider range of race conditions in client-side web applications.

The R4 tool from [55] was designed as a Stateless Model Checking tool of event-driven applications, which applied a dynamic partial order reduction(DPOR) technique to systematically explore the nondeterminism in the application and expose its overall effect.

we found

- R4 explored only 34 race conditions(*Total*),
- within the missing 16 race conditions from the 50, there are 5 harmful, 7 harmless and 4 bogus race conditions;
- within the 34 explored race conditions, R4 crashed when replaying 4 race conditions and reports 10 harmful races(High), among which 9 are indeed harmful, and 20 harmless races(Normal or Low), among which 19 are indeed harmful.

Altogether, R4 is able to report only 9 harmful race conditions among the 33 harmful races, while the rest 24 harmful races are either missing or misclassified by R4. However, R4 is mainly focusing on exploring the application instead of classifying race conditions; it detects race conditions not only from a single free run as EventRacer, but from all the explored interleavings using EventRacer. In other words, R4 can be a good source of input race-

condition warnings for our method.

4.6.2 Results on the Real Websites

Table 4.2 shows the result of our experiments on the websites of 70 randomly chosen companies on the Fortune-500 list, among which 20 websites do not have any *EventRacer* reported warnings; for the remaining 50 websites, the results are shown in the table. Columns 1-2 show the name and the number of *EventRacer* reported warnings (input). Column 3 shows the number of bogus warnings. Column 4 shows the number of harmful race conditions. Columns 5-8 show the number of harmless races and the number of undecided cases. Column 9 shows the total execution time. While using *EventRacer* to generate the input warnings, we found that some of the reported races do not make sense since they contain racing events with empty read/write accesses (likely due to defects in *EventRacer*). We have decided to filter them out before applying our classification method (labeled *Undetermined* in the table).

The results in Table 4.2 show that *RClassify* identified 132 harmful races out of 1,903 *EventRacer* reported warnings. We have manually inspected the state comparison results and confirmed the correctness of all these classifications. An example of these harmful race conditions is already shown in Figure 4.4, which turns out to be a real bug from www.newyorklife.com. While manually reviewing the classification results, we also noticed that harmful race conditions from the same website were often correlated in the sense that fixing one race condition (e.g., by adding a must-happen-before constraint) would fix many of the other race conditions.

The execution time of *RClassify* is largely determined by how fast the browser can fully load the target web page, since we always wait for the page to be completely loaded before recording the program state. In practice, it is common for a website to have large resource files such as JavaScript library code, which take a significant amount of time to download. An example is the website www.coach.com, which can take tens of seconds to load completely

Table 4.2: Experimental results on a set of real websites of Fortune-500 companies.

Name	Warning (input)	Bogus	Harmful	Harmless			Undetermined	Time
				console	b/c/t	none		
www.newyorklife.com	18	13	2	1	1	0	1	1m49s
www.aa.com	7	0	3	3	0	1	0	5m30s
www.adp.com	2	0	1	0	1	0	0	43.8s
www.yum.com	43	2	1	1	11	28	0	15m36s
www.simon.com	1	1	0	0	0	0	0	0
www.starbucks.com	8	2	1	0	4	1	0	1m38s
www.trw.com	8	0	0	0	3	5	0	2m48s
www.wdc.com	2	1	0	0	1	0	0	0
www.valero.com	36	2	8	1	23	2	0	6m41s
www.target.com	9	2	0	0	3	0	4	0
www.sjm.com	12	0	1	5	3	3	0	4m59s
www.disney.com	1	0	0	1	0	0	0	1m8s
www.honeywell.com	1	0	1	0	0	0	0	0m36s
www.coach.com	95	1	13	61	0	18	2	75m18s
www.unitedrentals.com	7	1	1	4	0	0	1	3m46s
www.buckeye.com	23	1	1	4	0	16	1	12m53s
www.biogen.com	26	1	0	6	0	4	15	7m40s
www.boeing.com	4	1	0	1	0	2	0	1m39s
www.cognizant.com	56	2	25	26	2	0	1	73m37s
www.heinz.com	3	1	0	2	0	0	0	1m22s
www.huntsman.com	70	2	18	0	0	45	5	21m18s
www.loews.com	4	3	1	0	0	0	0	37.6s
www.marathonoil.com	1	0	0	0	0	1	0	30.3s
www.mcdonalds.com	4	1	2	1	0	0	0	5m58s
www.regions.com	14	8	0	4	0	1	1	3m33s
www.bd.com	22	2	5	15	0	0	0	11m16s
www.campbellsoupcompany.com	5	3	1	1	0	0	0	1m16s
www.freddiemac.com	1	1	0	0	0	0	0	0
www.amazon.com/	10	6	0	0	3	0	1	0
www.generalcable.com	7	1	1	4	0	1	0	3m32s
www.l-3com.com	404	0	0	0	109	294	1	161m18s
www.nov.com	5	1	0	1	3	0	0	1m1s
www.sandisk.com	5	2	1	1	0	1	0	1m42s
www.wesco.com	5	1	3	1	0	0	0	2m3s
www.weyerhaeuser.com	29	0	21	8	0	0	0	14m45s
www.wd.com	2	1	0	0	1	0	0	0
www.unum.com	1	0	0	1	0	0	0	31.5s
www.altria.com/	13	2	3	2	4	1	1	4m5s
www.unitedhealthgroup.com	11	1	0	7	3	0	0	4m10s
www.unfi.com/	13	0	3	1	0	1	8	3m22s
www.tysonfoods.com	1	0	0	1	0	0	0	53.7s
www.pge.com	281	0	0	13	99	169	0	126m30s
www.tractorsupply.com	7	1	0	1	0	0	5	43.2s
www.wfscorp.com	16	1	0	8	0	6	1	7m33s
www.ally.com	91	0	8	71	0	8	4	67m48s
www.arrow.com	1	1	0	0	0	0	0	0
www.ashland.com/	6	0	0	0	1	5	0	3m17s
www.autozone.com/	9	4	1	0	2	0	2	41.6s
www.ball.com	2	0	0	2	0	0	0	1m14s
www.dollartree.com	501	0	6	14	238	243	0	235m31s
Total	**1,903**	73	**132**	273	515	856	54	

and thereby increase the total execution time of our tool. In addition, we intentionally inserted several sleep commands in our framework to ensure a smooth connection between operations, e.g., between the script for saving the state recording file and the script for running the next web page. During state comparison, we also run the same web application three times, among which two are in the original execution order while the third one is with the racing events flipped. For each individual replay, we observed only a slowdown of 1.5-2X when compared to a free run. Note that we have not optimized the runtime performance of our tool. Nevertheless, *RClassify* is fully automated and therefore is significantly more efficient than manually classifying race-condition warnings.

4.6.3 Compared to Heuristic Filtering

Table 4.3: Comparing *RClassify* with *EventRacer*'s filtering [108].

Total	EventRacer's high-risk	EventRacer's benign	EventRacer's the-rest
1,903	558	372	973
Harmful	**38** (6.8%)	**40** (10%)	**54** (1.8%)

We also compared the result of *RClassify* with the result of applying *EventRacer*'s heuristic filtering techniques, which do have the ability to check the effect of the racing events, but instead rely on the event type and a limited number of bug patterns. As such, they may miss truly harmful races as well as report many false positives. More specifically, *EventRacer* has two levels of filtering similar to ours. The first-level filtering aims at removing the *benign* races, and the second-level filtering aims at identifying the *high-risk* races. We use *the-rest* to denote the left-over races after these two levels of filtering. Table 4.3 shows our experimental results. Of the 1,903 race-condition warnings, *EventRacer*'s heuristic filtering techniques identified 558 as high-risk, 372 as benign, and 973 as the-rest. In contrast, our method showed that only 35 of the 558 high-risk races are actually harmful, whereas 40 of the 372 benign races are actually harmful. In addition, 57 of the remaining 973 races are actually harmful. These results show that such heuristic filtering techniques are not effective in practice.

Figure 4.7 shows a harmful race condition we detected from the website of Honeywell International. But *EventRacer*'s heuristic filtering considered it as *benign* (they call it *filtered*). The race condition is between the parsing event ev_a of asynchronous script *jsapi* and the parsing event ev_b of synchronous script *gssAutoComplete.js*. By re-executing the racing events in different orders, we have observed significant differences in many fields of the program states. One difference is in the title field of the web page: When $ev_a \prec ev_b$, the title is *"Home"*, whereas when $ev_b \prec ev_a$, the title is *"Honeywell - Global Technology Leader in Energy Efficiency, Clean Energy Generation, Safety & Security, and Globalization"*.

```
<-- other elements -->
<script id="script_4">
  ...
  var gjsapi = document.createElement('script');
  gjsapi.type = 'text/javascript';
  gjsapi.src = ('https:' == document.location.protocol ? 'https://' : 'http://')
      + 'www.google.com/jsapi';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(gjsapi, s);
  ...
</script>
<-- other elements -->
<script type="text/javascript" id="script_26" src="http://honeywell.com/_layouts/
InternetFramework/Scripts/gssAutoComplete.js"></script>
<-- other elements -->
```

Figure 4.7: A race EventRacer filtered but it's indeed harmful.

4.6.4 Limitations

Although our method is effective in classifying most of the *EventRacer* reported warnings and is scalable for practical use, there are still some limitations. The first limitation is that we can only control the racing events that execute JavaScript at the callback function level, as opposed to the level of individual statements in the JavaScript code. As such, we have to treat each callback function as a black box, and therefore lack the control and data de-

pendency relations over shared variables. This can sometimes prevent us from classifying race conditions with more accuracy. The second limitation is that, since we store the instrumented web sites in a local web server and make sure the input HTML and resource files remain the same during different executions, the behavior of the web application may be slightly different from the original one, e.g., when the client-side application needs to have meaningful interactions with databases in the server. As such, we can only classify race conditions over shared objects in the memory at the client side. Currently, we cannot handle races that cause differences in databases on the server side.

4.7 Related Work

There are a number of tools for detecting race conditions in web applications, some of which are based on static analysis [163] while others are based on dynamic analysis [108, 101, 52]. Static analysis has the advantage of covering all possible execution paths of the JavaScript code. However, due to the rich set of dynamic features in JavaScript, such as *eval*, callback, and dynamic creation of code, web applications based on JavaScript are known to be difficult to analyze statically [45, 57, 110, 124, 87, 157, 126]. Dynamic analysis tools such as *EventRacer* [108, 101] and WAVE [52] do not have such limitations. However, dynamic analysis does not have good code coverage [11, 146, 148, 147, 149] and therefore can miss bugs; they can also report many bogus warnings. Furthermore, none of these existing methods classifies race conditions based on replay and state comparison.

Mutlu et al. [95, 94] proposed a method for detecting race conditions using a combination of dynamic and trace-based static analysis. That is, they first leverage dynamic analysis to obtain an execution trace and then apply a form of predictive static analysis to the trace, to detect races that may occur in all possible interleavings of the events in the given trace. However, our method differs from their method significantly. First, their method does not use replay to check the real impact of the racing events and therefore may still report bogus and

harmless races. Second, their method seems to be limited to races between AJAX requests and races over persistent state such as the *cookie*. In contrast, our method can handle a wider range of race conditions and is also more accurate, as shown by our experimental results.

Jensen et al. [55] proposed a stateless model checking method for exploring the nondeterminism in the application and concisely exposing its overall effect, which is useful for bug discovery. More specifically, they adopt a dynamic partial order reduction (DPOR) technique for reducing the search space, and systematically explore the nondeterminism by reordering certain events. Unlike race detecting methods such as EventRacer [108, 101], that detect race conditions only on one trace, this method detects race condition on all explored traces, thus more race condition can be discovered(e.g., race conditions only appear only when certain events are reordered). However, our method is fundamentally different in that it relies on deterministic replay and has higher accuracy in classification. As a matter of fact, their method can serve as a good replacement of EventRacer, such that more race conditions can be reported, including those are hard to expose from a free run of the web application.

There are also prior works on deterministic replay for web applications [89, 9, 18], but their goal is to allow developers to revisit any recorded program state to diagnose bugs. There are also some existing methods for dynamically migrating running web applications [15, 81, 80], but their focus is on transferring a live session of the running application from one browser to another browser or from one device to another device. In this context, they also need to address the problem of recording of program states as well. However, the purpose of recording states is significantly different from ours. In particular, none of the prior works on state recording and replay focuses on diagnosing concurrency bugs.

For the closely related problem of detecting and analyzing *data-races* in multithreaded programs, there is a large body of existing work [96, 114, 137, 66, 98, 123, 111, 159]. Specifically, Narayanasamy et al. [96] proposed perhaps the first replay-based method for classifying data-races in multithreaded C/C++ applications. They employ a checkpointing tool to take snapshots of the main memory during the execution of the program, and then compare the

snapshots to decide if a data-race is harmful. Similar works also include Koushik Sen's race-directed random testing tool [114] and the *Portend* tool by Kasikci et al. [66]. There also exist software testing tools such as CTrigger [98], PENELOPE [123] and Maple [159], which first identify the potentially buggy thread interleavings and then replay these interleavings to see if they can cause program failures.

However, detecting and classifying traditional *data-races* in multithreaded C/C++ or Java programs are different from detecting and classifying *race conditions* in client-side web applications. The computing platforms are significantly different in that one relies on multithreading whereas the other relies on event-driven programming. The supporting tools are also different: For multithreaded programs, there exist a large number of checkpointing tools, but for client-side web applications, we are not aware of any such tool. Due to these reasons, both deterministic replay and program-state comparison require drastically different solutions. Therefore, except for the high-level similarity, *RClassify* is completely different from the existing methods and tools.

4.8 Discussions

We have presented the first replay-based method for automatically classifying race-condition warnings in client-side web applications. It is based on *re-executing* racing events in different orders and then *comparing* the resulting program states. We have developed a purely JavaScript-based framework for monitoring and controlling the execution order of racing events. We have also developed a method for recording and comparing relevant fields of the program states. We report a race condition as being harmful only if the execution order of the racing events affects relevant fields of the program states. We have implemented our method and evaluated it on a set of standard benchmarks as well as many real-world websites. Our experimental results show that the method is effective and scalable for practical use.

Chapter 5

Repairing Race conditions in Web Applications

Modern web applications are becoming increasingly complex due to the need to implement a large number of features while ensuring a speedy response to users. As a result, there is a pervasive use of JavaScript and its many dynamic language features. This event-driven asynchronous programming framework is prone to concurrency bugs. Although web browsers guarantee to execute JavaScript in a single-threaded manner, thus avoiding *data-races*, there can still be many high-level race conditions as a result of interleaving the various event handlers, asynchronous parsing of HTML elements, timer events, Ajax requests, and their callback routines.

A race condition consists of two *unordered* events that are accessing the same resource, and at least one of the event is a write operation. A race condition is harmful if executing the two events in different orders can result in different behaviors, one of which is expected while the other is not. Unexpected behaviors include exceptions, wrong results, and unresponsive interfaces, all of which can decrease user satisfaction and potentially damage the reputation of the organization.

```
<button id = "b1" onclick = "fn()"> Button </button>
<!-- other elements -->
<script id = "s1">
  function fn(){...};
</script>
```

Figure 5.1: An example race condition in web application.

Figure 5.1 shows a race condition between the firing of *b1.onclick*, whose callback function is *fn*, and the parsing of the script that defines *fn*. This race is harmful because a user may click the button before *fn* is defined, leading to a *ReferenceError* exception. Although in principle, eliminating this type of bug is straightforward, e.g., by avoiding the execution of *fn* unless it is defined, in practice, this is not always easy to implement. For example, there may be several ways of imposing the ordering constraint: we may move the script before the button tag, move the button tag after the script, or disable the clicking until *fn* is defined. Each solution may or may not be feasible depending on the dependencies between these two events and the other HTML elements. In addition, performance overhead may vary for different repairs.

We propose the first dynamic analysis-based method for automatically diagnosing such race conditions and computing efficient repairs. Although in recent years various tools have been proposed for detecting race conditions in web applications [101, 108, 52, 95, 56], none of these tools can identify the truly harmful race conditions and compute the repairs. Wang et al. [140] developed a static analysis-based tool to compute repairs for racing events that do not involve dynamic features. However, since it relies on static analysis only, it cannot handle race conditions in applications where the use of dynamic features is pervasive. In contrast, our new method, named *RCrepair*, leverages dynamic analysis to robustly diagnose these race conditions and compute the repairs.

Static analysis of web applications is difficult because a variable in JavaScript may hold a primitive number, a string, or a pointer to another variable, function, or DOM element. For example, *"image1"* in Figure 5.2 could be the result of dynamically concatenating *"image"*

```
1  <script>
2    function foo(index){
3      var s1 = "image" + index;
4      var s2 = "lo";
5          s2 += "ad";
6      document.getElementById(s1)
7        .addEventListener(s2,function(){...},false);
8    }
9    function bar(){
10     document.getElementById("image1")
11       .addEventListener("load",function(){...},false);
12   }
13   ...
14   foo(2);
15 </script>
```

Figure 5.2: JavaScript code using dynamic features.

and *index*, while *load* is the result of concatenating *"lo"* and *"ad"*, by invoking *foo(1)* during the execution of some other JavaScript code. If the *image1.onload* event is involved in a race condition, tracing it back to the handler added in Lines 6-7 would be difficult using static analysis only. Consequently, it is difficult for such static analysis tools to compute the repair, because modifying the function *foo* may not even be allowed depending on the use of the function in other places, e.g., *foo(2)* in Line 14.

In addition to the dynamic features of JavaScript, web applications often rely on third-party libraries such as *jQuery*. In fact, nearly every official website of the current Fortune-500 companies uses third-party libraries, many of which are heavily optimized and often obfuscated. Of the 42 real websites we studied during our experiments, 32 of them used *jQuery*. Although some of the existing approaches advocate the use of manually specified behavioral models during static analysis [140], they require the developer to have a deep understanding of the APIs in these libraries, which is tedious and error-prone. Furthermore, third-party libraries are constantly being updated—*jQuery*, in particular, has more than 140 versions in use. Therefore, in practice, tools built upon such static analysis techniques tend to be fragile.

In contrast, *RCrepair* relies on dynamic analysis and therefore can avoid the aforementioned problem. Specifically, it directly monitors the execution of the JavaScript code to more effectively identify the racing events, regardless of how complex the JavaScript code is or whether it involves the use of third-party libraries.

The repairs computed by our method fall into three categories, labeled as Type 1, Type 2, and Type 3 repairs, respectively. Type 1 repairs rely on reordering the HTML tags. That is, we eliminate the unexpected event order by swapping the racing elements in the HTML file. Type 2 repairs rely on adding simple conditional statements, e.g., to check whether the function *fn* is defined before invoking it. Type 3 repairs, in contrast, rely on delaying the execution of a racing event at run time with periodic retry. That is, we keep postponing the event using *setTimeOut* until it is ready to execute. Toward this end, we develop a new deterministic replay framework to decide, for each race condition, what the desired execution order is and what type of repair is the most efficient.

Our replay framework is platform agnostic in that we transform the HTML file of the application to add self-control capabilities, as opposed to modifying the underlying web browser. Since web browsers are constantly updated, tools implemented for a particular version of a web browser will become obsolete quickly. In contrast, our platform-agnostic approach is more robust against changes and updates. For each pair (ev_a, ev_b) of racing events, we first execute the application while forcing ev_a to occur before ev_b, and then execute the application while forcing ev_b to occur before ev_a. We say the race condition is real, and is harmful, only if both executions are feasible and they result in different program states. Here, a program state is defined as the content stored in the HTML DOM, JavaScript variables, and environment variables inside the web browser.

We have implemented the *RCrepair* tool and evaluated it on both standard benchmarks for web application testing [101, 108, 52] and real websites from the current Fortune-500 companies. Out of the 500 websites we studied, there are 122 harmful race conditions detected from 42 websites—they serve as the starting point of our evaluation. After applying

RCrepair, all of the 122 race conditions were successfully repaired. Among them, 7 were eliminated by Type 1 repairs, 57 were eliminated by Type 2 repairs, and 56 were repaired by Type 3 repairs. Furthermore, the time taken to compute the repairs was typically a few seconds, with runtime overhead of less than 5%, indicating that *RCrepair* can handle real websites.

To sum up, we make the following contributions:

- We propose the first dynamic analysis-based method for automatically repairing race conditions in JavaScript-based web applications.
- We implement the *RCrepair* tool and evaluated it on a large number of real websites.
- We show empirical evidence that the tool is both effective in repairing race conditions and efficient for practical use.

5.1 Motivating Examples

Consider the web page in Figure 5.3, which has an image tag, a JavaScript code block, and a text element named *outputField*. The *image.onload* event, fired after the browser downloads *image1*, invokes the function *image1Loaded()* to change the text in *outputField* to `'Done!'`. Therefore, the expected sequence of events is ev_1*: parsing(image1)* $\rightarrow$ ev_2*: parsing(script1)* $\rightarrow$ ev_3*: parsing (outputField)* $\rightarrow$ ev_4*: firing(image1.onload)* However, depending on the network speed, there can be two race conditions:

1. **RC1** is (ev_2, ev_4) over *image1Loaded*
   (a) event ev_2: parsing(script1)
   (b) event ev_4: firing(image1.onload)
2. **RC2** is (ev_3, ev_4) over *outputField*
   (a) event ev_3: parsing(outputField)

```
<img src = "image1.jpg" onload = "image1Loaded('Done!')" id = "image1">
<script id = "script1">
  function image1Loaded(message) {
    document.getElementById("outputField")
    .innerHTML = message;
  }
  ...
</script>
<!-- other elements -->
<div id = "outputField"></div>
```

Figure 5.3: A web page with two harmful race conditions.

(b) event ev_4: firing(image1.onload)

RC1 is a race between the parsing of *script1* and the firing of *image1.onload*. Typically, the parsing event finishes first, but if *image1* is downloaded before the parsing finishes, e.g., due to slow parsing of other HTML elements preceding *script1*, *image1Loaded* will be undefined when the browser invokes it, causing a *ReferenceError* exception.

RC2 is a race between the parsing of *outputField* and the firing of *image1.onload*. Even if RC1 does not occur, *image1.onload* may still be fired before *outputField*, causing *document.getElementById ("outputField")* to throw an exception due to the undefined *outputField*.

The main idea for eliminating these race conditions is to impose a happen-before constraint to disallow the unexpected execution order. For example, **RC1** may be eliminated by moving *script1* before *image1* in the HTML file.

As for **RC2**, we may not be able to swap the order of *image1* and *outputField* due to potential dependencies with JavaScript code blocks located between *script1* and *outputField*. We cannot move *outputField* upward either because the order of visible HTML elements determines the layout, which is not supposed to be changed. Therefore, our solution in this case is to wrap up the original callback function of *image1.onload* and delay it using repeated calls to *setTimeout*, until the callback function is ready to execute.

```
<script id = "script1">
  function image1Loaded(message) {
    document.getElementById("outputField")
    .innerHTML = message;
  }
  ...
  </script>
  <script>
    function image1Loaded_wrapper() {
    if (!document.getElementById("outputField"))
      setTimeout(image1Loaded_helper, 50);
    else
      image1Loaded('Well done!');
    }
</script>
<img src = "image1.jpg" onload = "image1Loaded_wrapper()" id = "image1">
<!-- other elements -->
<div id = "outputField"></div>
```

Figure 5.4: Repairs of **RC1** and **RC2** from Figure 5.3.

The repaired version is shown in Figure 5.4, where *image1.onload* is replaced by *image1Loaded_wrapper*. The wrapper function repeatedly checks if *outputField* exists and invokes *image1Loaded* only after *outputField* is defined. This is consistent with what developers would do in practice to implement customized synchronization.

Also note that, in this example, the non-intrusive use of helper function is better than directly modifying the JavaScript code in *image1Loaded*, because the function may also be used as the callback function of other events; in such cases, modifying it directly would change the behavior of the application.

In terms of runtime overhead, it is easy to see that the repair for RC1 does not introduce any overhead, whereas the repair for RC2 introduces a modest overhead as a result of the timer-based synchronization. However, it is necessary for repairing the race condition.

5.2 Algorithm

In this section, we present our high-level algorithm for repairing race conditions.

5.2.1 The Scope

We focus on repairing race conditions between events in client-side web applications. A modern HTML page consists of a tree of elements, each of which has an opening and a closing tag, e.g., `<p>...</p>`. Elements are declared either statically in the page or dynamically by executing some JavaScript code. The document object model (DOM) is a tree representation of the HTML elements to be rendered by a browser. Each node in the DOM tree may have attributes to hold meta-data. JavaScript code, by default, are synchronous. However, when declared using the "async" or "defer" attribute, a deferred script will run after all static HTML elements in the web page are parsed, while an asynchronous script may run at any time after it is downloaded by the browser.

Web applications are written in an event-driven programming style, where various handlers are registered to DOM nodes to react to user interactions and DOM changes, such as *onclick* and *onload*. Since the browser ensures that each JavaScript code block is executed in a single-threaded fashion, there is no *data-races* in the traditional sense (unlike in multithreaded Java/C++ programs). However, there can still be *race conditions* among event handlers.

Let $\rightarrow_{mhb}$ be the must-happen-before relation over events such that $(ev_a, ev_b) \in \rightarrow_{mhb}$ if and only if ev_a precedes ev_b in all possible executions of the application. Let $\rightarrow$ be a happens-before relation such that, if ev_a precedes ev_b in one execution, we say $ev_a \rightarrow ev_b$. Given an event ev, we use $WR(ev)$ to denote the set of memory locations written by ev, and $RD(ev)$ to denote the set of memory locations written by ev. A race condition is defined as a pair (ev_a, ev_b) of events such that

1. $(ev_a, ev_b) \notin \rightarrow_{mhb}$, $(ev_b, ev_a) \notin \rightarrow_{mhb}$, and

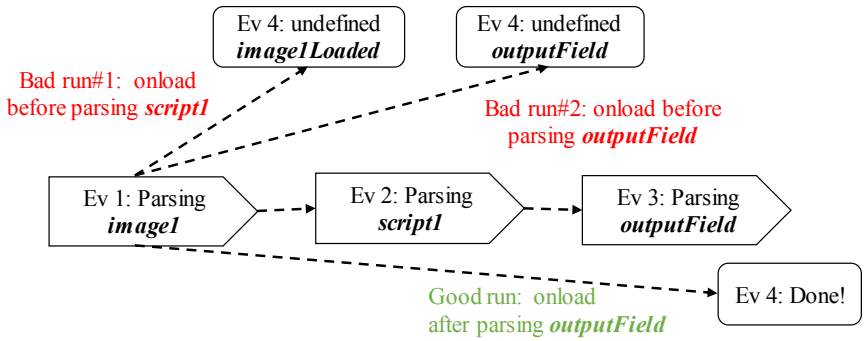


Figure 5.5: Three executions for the application in Figure 5.3.

2. $\exists var \in WR(ev_a) \cap WR(ev_b)$ or $var \in WR(ev_a) \cap RD(ev_b)$ or $var \in RD(ev_a) \cap WR(ev_b)$.

In Figure 5.5, for example, the dashed lines denote the $\rightarrow_{mhb}$ relation for the application in Figure 5.3. Since there is no edge between ev_2 and ev_4 or between ev_3 and ev_4, we know that ev_4 is racing with ev_2 and ev_3.

However, in practice, not all race conditions are harmful. Although there exist tools for detecting race conditions in web applications [101, 108, 52, 94, 95], none of them can robustly separate the harmful races from the harmless ones. In this work, we develop a platform-agnostic *deterministic replay* framework to check, among other things, whether a race condition is indeed harmful. We say the race (ev_a, ev_b) is harmful if and only if both $ev_a \rightarrow ev_b$ and $ev_b \rightarrow ev_a$ are feasible, and they may result in different program states. This new deterministic replay framework also serves as the foundation for implementing our repair algorithm.

5.2.2 The Overall Procedure

The overall flow of *RCrepair* is presented in Algorithm 8, which takes the HTML file and the two racing events ev_a, ev_b as input, and returns the modified HTML file as output. The racing events may be generated using any of the existing race detection tools.

First, we invoke the subroutine PreRepair to perform an automated analysis of the given

Algorithm 8 The Overall Repair Algorithm.

```
 1: PreRepair ( HTML, ev_a, ev_b ) {
 2:     s_1 ← Replay (HTML, ev_a → ev_b)
 3:     s_2 ← Replay (HTML, ev_b → ev_a)
 4:     assert ( both s_1 and s_2 exist ∧ s_1 ≠ s_2 );
 5: }
 6: Repair ( HTML, ev_a → ev_b ) {
 7:     HTML' ← repair_type1 (HTML, ev_a → ev_b)
 8:     if (HTML' does not exist)
 9:         HTML' ← repair_type2 (HTML, ev_a → ev_b)
10:     if (HTML' does not exist)
11:         HTML' ← repair_type3 (HTML, ev_a → ev_b)
12:     return HTML';
13: }
14: PostRepair ( HTML', ev_a, ev_b ) {
15:     s_1 ← Replay (HTML', ev_a → ev_b)
16:     s_2 ← Replay (HTML', ev_b → ev_a)
17:     assert ( s_1 exists ∧ s_2 does not exist );
18: }
```

race using deterministic replay. For each race condition (ev_a, ev_b), we run the application twice. In the first run, we will enforce the execution order by forcing ev_a to happen before ev_b. While in the second run, we swap the execution order of the two events. By checking that both executions are feasible and they lead to different program states, we make sure the race is indeed harmful.

The reason why PreRepair is needed is because existing race detection tools for web applications often return many false positives, due to their limitations in modeling the happens-before relations [101, 108, 52]. Although some of these races may be harmful, the vast majority are not, which means repairing them would be counter-productive.

After making sure that the race is indeed harmful, the next step is to decide which of the two execution orders is desired. If deterministic replay triggers an exception, for example, the corresponding execution order is obviously undesired. In general, however, manual intervention from the developer is unavoidable because, ultimately, only the developer can decide what is the desired program behavior. However, our tool also automatically infers the developer's intent and provides suggested ordering constraints based on heuristic rules. The inferred ordering constraints can be reviewed by the developer, to see if they are desired. Specifically,

we follow the following rules to infer the likely order. For a race condition (ev_a, ev_b), if

- ev_a is a parsing event of an element in the main HTML, then $ev_a \rightarrow ev_b$;
- ev_a is a passive (user-triggered) event, then $ev_a \rightarrow ev_b$;
- ev_a is the first-def of the read operation ev_b, and there is no def/use between ev_a and ev_b, then $ev_a \rightarrow ev_b$;
- when none of the above rules can be applied, we infer the likely order by observing the runtime results. That is, in a free run, if ev_a appears before ev_b more often, then $ev_a \rightarrow ev_b$.

Next, we invoke REPAIR to compute the repair. It takes the HTML file and the happens-before relation over racing events as input and returns the modified HTML file as output. Internally, it first checks whether the happens-before relation can be enforced by reordering the racing HTML elements – this is called the Type 1 repair. If this is not possible, it checks whether the happens-before relation can be enforced by adding conditional statements – this is called the Type 2 repair. If neither repair type is applicable, it adopts the Type 3 repair by intercepting the racing events at run time and imposing the happens-before relation using customized synchronization.

Finally, we invoke the subroutine POSTREPAIR to confirm, through deterministic replay, that the race condition indeed has been eliminated.

5.3 Detailed Algorithms

We present the repair strategies in more details.

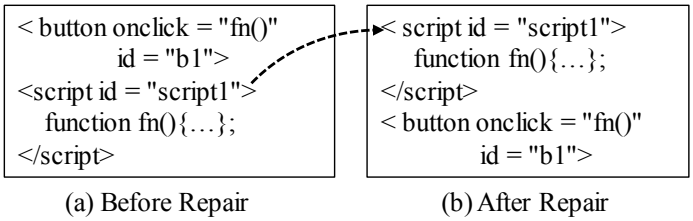


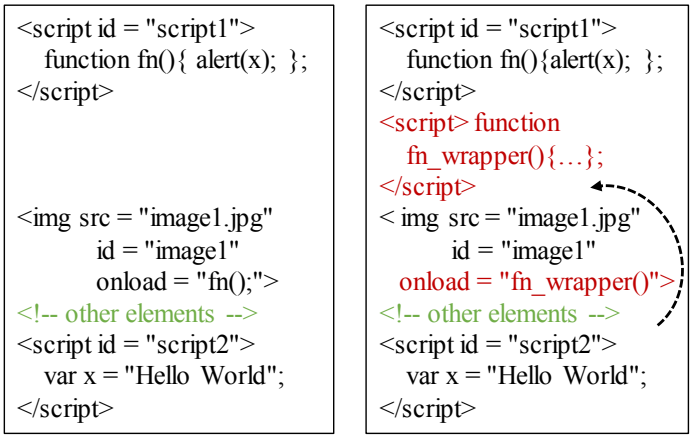
Figure 5.6: Example for Type 1 repair (reordering).

5.3.1 Three Repair Strategies

Recall that we perform the repair in three steps, which correspond to the Type 1, 2, and 3 repairs, respectively. Type 1 repairs are implemented by changing the order of racing elements in the HTML file. Type 2 repairs are implemented by adding conditional statements in the HTML file. Whereas Type 3 repairs are implemented by preloading a JavaScript library to add self-monitoring and self-control capabilities to the application; at run time, the racing events are intercepted and postponed to enforce the desired happens-before relation. Details of these three repair types are presented as follows.

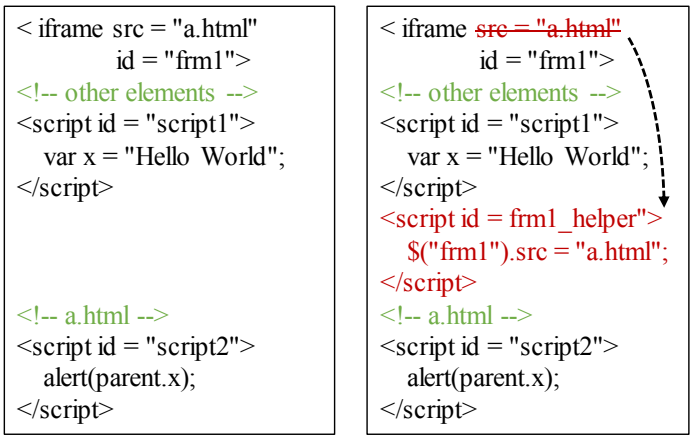
Type 1 Repair The most straightforward way of eliminating a race condition is to reorder the corresponding HTML elements. For example, the race in Figure 5.6 is between parsing *script1* and *b1.onclick*. Our repair is to reorder the button and the script in the HTML to make sure *fn* is defined before the button is clicked. However, to avoid introducing new bugs, we need to check if there is a cyclic dependency between the two elements, or between them and elements located in between. To avoid complications of static analysis, we check *syntactically* if there are JavaScript code blocks (or HTML files that may contain JavaScript) between the two racing elements. If there is, we do not perform Type 1 repair since hidden dependencies may arise at run time.

Type 2A Repair When Type 1 repair is not applicable, we modify the HTML slightly to eliminate the race condition. One such modification is Type 2A repair, which is well suited



(a) Before Repair (b) After Repair

Figure 5.7: Example for Type 2A repair (onload).

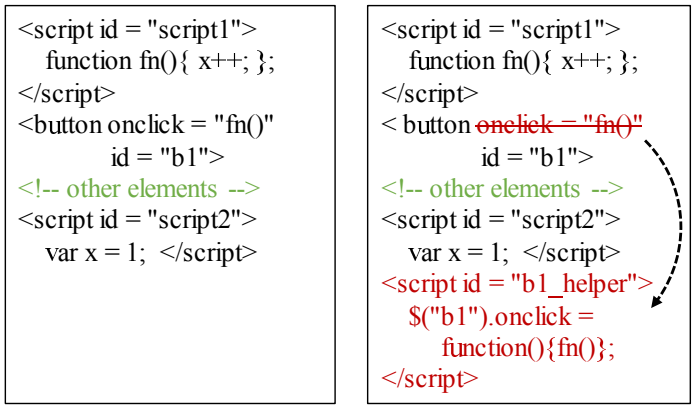


(a) Before Repair (b) After Repair

Figure 5.8: Example for Type 2A repair (async).

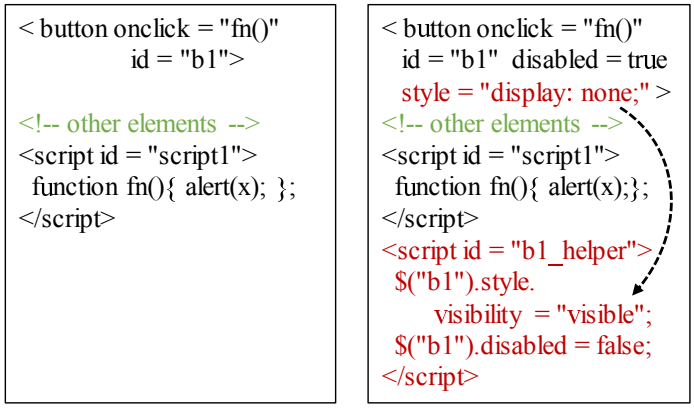
for handling either the loading of asynchronous resource or its onload event handler.

- Figure 5.7 shows the example of an *onload* event handler. Our solution is to wrap up the callback function *fn* using customized synchronization. That is, *fn_wrapper* uses *setTimeOut* to postpone the execution of *fn* until the function is defined.
- Figure 5.8 shows the example of asynchronous loading. Our solution is to delay the loading of the resource by removing the *src* attribute of the HTML element initially, and adding it back only after the other racing event ends.



(a) Before Repair (b) After Repair

Figure 5.9: Example for Type 2B repair (onclick).



(a) Before Repair (b) After Repair

Figure 5.10: Example for Type 2B repair (disabled).

A common bug in real web applications is when the *onload* event handler is added asynchronously by another script, but the addition occurs after the resource is loaded, and thus the *onload* will not be fired. It is worth noting that such bugs are difficult to catch using static analysis, but we can still fix it using the above approach.

Type 2B Repair While Type 2A targets spontaneous events, Type 2B targets passive (user-triggered) events. In Type 2B repair, we delay the addition of passive event handlers or temporarily disallow users to interact with the element.

```
<script id = "script1">
  var zz = 1;
  function fn() {
    zz = 2;
    xx ++;
    zz = 0;
  }
</script>
<button id = "b1" onclick = "fn()">Button 1</button>
<script id = "script2">
  var xx = 1;
</script>
```

Figure 5.11: Half-executed JavaScript code.

- Figure 5.9 shows the example of delaying the addition of an *onclick* event handler. It is suitable when the event handler tries to access a shared resource that has not yet been defined, in which case the browser would throw an exception and stop executing the current JavaScript at the exception location, causing half-executed JavaScript code and often leading to bugs. The repair is to delay the addition of the event handler until such shared resource is defined.

  In practice, it is common for developers to add interactive event handlers when the web page is loaded, to prevent them from accessing resources that are not yet ready. For example, in Figure 5.11, if the button *b1* is clicked before *script2* is parsed, when *fn* reaches the statement *xx ++;*, it will throw an exception. Since the code before *zz = 2;* will be executed but the code after *zz = 0;* will not, it may cause the application to fail later on.

- Figure 5.10 shows an example for repairing race conditions related to passive (user-triggered) event handlers. This is also a common bug pattern in practice, where the event handler is defined but the script that contains the definition of the callback function is not executed yet. And delaying the adding of the event handler will not help to prevent the unwanted behavior(Also see Figure5.20). Our solution is to prevent the

user to interact with the element, consequently prevent the frustration from failing to interact with the element - by temporarily disabling and making the element invisible.

This is well suited for race conditions that allow the user to interact with an element before its handlers are attached. Alternatively, if the event handler tries to access a shared resource that has not yet been defined, the browser would throw an exception. Our solution for both is to temporarily remove the event handler, and re-attach it only after the racing event ends, as shown for the example in Figure 5.9.

Type 3 Repair While Type 1 and 2 repairs are efficient at runtime, there are race conditions that cannot be eliminated by either approach. Such hard-to-repair race conditions usually involve the dynamic features of JavaScript and event-driven environment (e.g., timer, Ajax, dynamically generated DOM node). Due to the aforementioned limitations, these racing events are difficult to locate and control statically. In such cases, we resort to Type 3 repairs, where we enforce the desired happen-before relation by intercepting the racing events at run time and controlling their execution order. Although in principle we can also use this approach to repair races that can be fixed by Type 1 and Type 2 repairs, it would have been less efficient. Thus, we always check if Type 1 and Type 2 repairs are applicable before applying Type 3 repairs.

RC1 in Figure 5.3 can be fixed by Type 1 repair. Another way to fix it is using Type 2 repair, which delays the setting of the *src* attribute of the image. In contrast, Type 3 repair would add a helper script to set the flag after the parsing of *script*. Then, the wrapper function for *image1.onload* can check if it is ready.

Type 3 repair works as follows. First, we check if each of the two racing events can be instrumented statically. In Figure 5.12, for example, ev_2 and ev_3 can be instrumented statically by adding a helper script to signal the end of their parsing, while ev_4 can be instrumented statically by replacing the callback function with a wrapper function as described before. Specifically, the condition becomes *if (!finish_waiting("image1_onload"))* , where the *finish_waiting*

```
<script>
  function image1Loaded_wrapper() {
    if (!finish_waiting("image1_onload"))
      setTimeout(image1Loaded_helper, 50);
    else
      image1Loaded('Well done!');
      ev_executed("image1_onload");
    }
</script>
<img src = "image1.jpg" onload = "image1Loaded_wrapper()" id = "image1">
<script id = "script1">
    function image1Loaded(message) {
      document.getElementById("outputField")
      .innerHTML = message;
    }
    ...
<script>
<script> ev_executed("script1_parsed"); </script>
<!-- other elements -->
<div id = "outputField"></div>
<script> ev_executed("outputField_parsed"); </script>
```

Figure 5.12: Repairs of **RC1** and **RC2** from Figure 5.3.

and *ev_executed* functions are defined inside *RCrepair.js*, and the must-happen-before relation is defined by *RaceInfo.js*. The *finish_waiting(ev)* function checks whether *ev* has finished waiting. In this example, it will check whether both *script1_parsed* and *outputField_parsed* have finished executing.

After parsing the race information, *RCrepair.js* will create a *wait_list* for each racing event, containing the set of events for which it needs to wait. In the above example, for instance, both *wait_list("outputField_parsed")* and *wait_list("script1_parsed")* are empty, whereas we have *wait_list("image1_onload")* = {*"outputField_parsed","script1_parsed"* }.

Figure 5.13 shows an example of the Type 3 repair, where two customized JavaScript libraries are loaded before the rest of the HTML file is parsed. They intercept the event handler registration APIs to create a helper function for each callback function at run time. As shown in Figure 5.14, each *fn_wrapper* function repeatedly checks the value of *finish_waiting(ev)* to

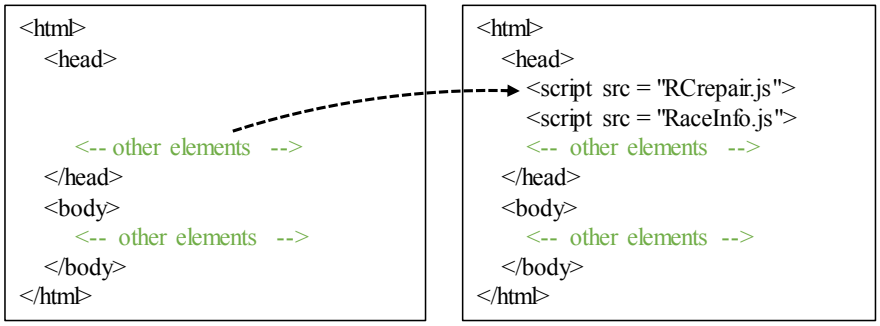


Figure 5.13: Example for Type 3 repair: Dynamically intercepting events and imposing the happens-before order.

decide if all events upon which *fn* waits have ended. Since JavaScript does not have native synchronization primitives, this timer-retry design pattern is widely used by developers in practice.

```
<script>
  function fn_wrapper() {
    if (!finish_waiting(ev))
      setTimeout(fn_wrapper, 50);
    else
      fn(arguments);
      ev_executed(ev);
  }
</script>
```

Figure 5.14: The *timer-retry* function for Type 3 repair.

5.3.2 Intercepting Racing Events

When the racing events can be located in the HTML file, we instrument it statically using source-to-source transformation. When the racing events cannot be located statically, we intercept it at run time.

Prior to the parsing of the HTML file, we load a customized JavaScript library named *RCrepair.js* as well information of the racing events. The execution of *RCrepair.js* re-defines native

JavaScript APIs that are related to generating events, including the registration and removal of event handlers. The *RCrepair.js* library contains instrumented APIs for replacing a small subset of JavaScript native APIs to add *wait-notify* style synchronizations. In JavaScript, each DOM element or JavaScript object inherits the fields and APIs of its *prototype* when the object is created. By loading *RCrepair.js*, we replace the event-handler-related APIs with our instrumented version. Later on, whenever a new object is created, our instrumented API will be inherited. Although third-party JavaScript libraries often define their own APIs for adding/removing event handlers, internally, these third-party APIs still rely on the small set of native JavaScript APIs. Thus, they will also be handled in the same fashion. *RCrepair.js* also contains the utility functions such as *finish_waiting()* and *ev_executed()*.

Prior to the parsing of the HTML file, we also load another customized JavaScript library named *RaceInfo.js*. This library contains information of the race events to be repaired at runtime. Each racing event is a tuple that contains:

- The event type, such as *active_eh*;
- The element id(if applicable), such as *image1*;
- The event handler type(if applicable), such as *onload*;
- Is the event defined statically? For example, the *image1.onload* event handler in Figure 5.3 is static;
- Is the event a read or write operation of the racing variable?
- The line of code in the HTML;
- The JavaScript code executed in this event (optional), if the event cannot be identified by any of the above methods, such as timer events.

We have modified EventRacer to automatically generate the race information. However, in practice, one may use any technique to generate the race information, including creating it manually, as long as the information is represented in the above format.

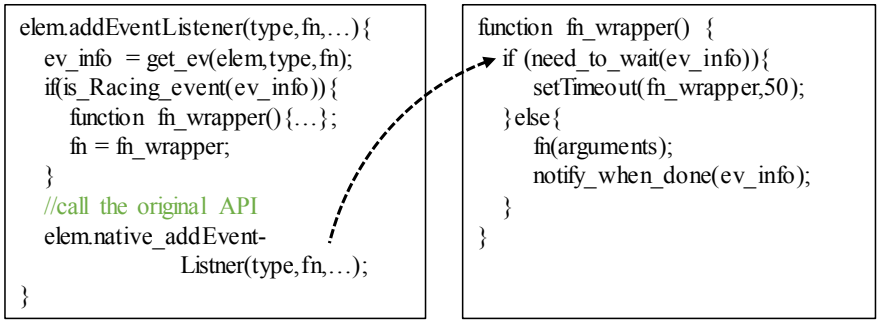


Figure 5.15: Handling dynamic events in Type 3 repair (callback function).

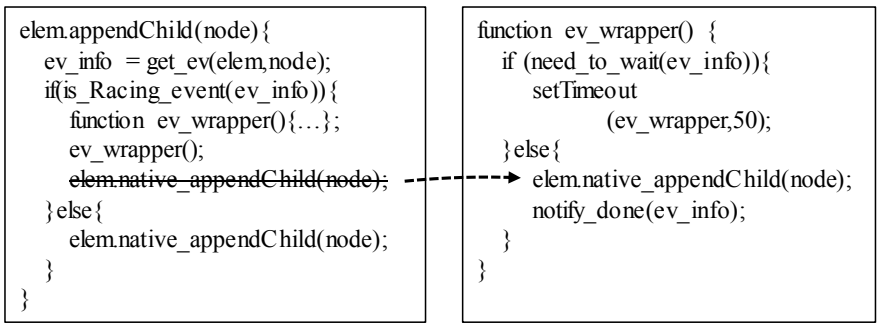


Figure 5.16: Handling dynamic events in Type 3 repair (the event itself).

Figure 5.15 shows the dynamic instrumentation of a callback function named *fn*, whereas Figure 5.16 shows the dynamic instrumentation of an asynchronous loading event. In the latter case, the original JavaScript tries to insert a dynamically created DOM node. Instead of inserting the DOM node directly, our instrumentation tool calls a timer-based wrapper, which periodically checks if the other racing event has ended before inserting the DOM node.

Note that *appendChild()* in Figure 5.16 (left) has been replaced by our own version, defined in *RCrepair.js*, which calls the original API eventually, but before that, checks if it is racing with another event. If it is not racing with another event, the original API will be called. But if it is racing with another event, we will create a wrapper function to periodically check if the other event has ended.

Depending on the event types, the wrapper function is created in one of the following two

ways:

- When the racing event is an asynchronous callback, including the callback of timer, Ajax *onreadystatechange* event handler, and the spontaneous event handlers, our method will wrap the callback function and replace the call to the original function with the call to the generated wrapper function.
- When the racing event is the action itself, including DOM node insertion, asynchronous resource loading, sending Ajax request, and passive event, our wrapper function will be used to delay the event. We include passive (user triggered) event handler here to prevent the user interaction before the element is ready.

5.4 Experiments

We have implemented our new method in a software tool named *RCrepair*, which builds upon a number of open-source software. Our HTML parsing and source-code transformation component was implemented using *jsoup 1.8.1* with 3.5K lines of our own Java code. Our runtime monitoring and control component was implemented as a library (*RCrepair.js*) with 1K lines of JavaScript code. Our deterministic replay component was implemented with 2.4K lines of Java code and 2.1K lines of JavaScript codes. During the experimental evaluation, we stored all benchmarks under test in a local web server. We used *Teleport Ultra 1.69* to download the source files and *Mozilla Firefox 40.0* as the browser. All experiments were conducted on a PC with Intel Core i7-4700 2.4 GHz CPU running 64-bit Ubuntu 12.04, with 4GB RAM.

5.4.1 Benchmarks

The benchmarks used in our experiments fall into two groups. The first group consists of manually created web applications from recent publications such as WebRacer [101], Even-

tRacer [108] and WAVE [52]. Since these benchmarks are hand-crafted specifically for illustrating the most common race condition patterns, the majority of them are known to be harmful. Our goal is to use these benchmarks to confirm that *RCrepair* can correctly repair the common race conditions. The second group consists of the official websites of the current of Fortune-500 companies. We leverage the EventRacer tool to generate the initial list of race conditions for all the 500 websites. After applying the deterministic replay-based classification, we found a total of 122 harmful race conditions from 42 websites. For each of these 122 race conditions, we confirmed that both of the two execution orders are feasible (the race is real) and they led to different program states (the race is harmful). We manually inspected them to confirm that repair was indeed needed. This served as the starting point of our experimental evaluation. Our goal is to decide if *RCrepair* can efficiently repair these real bugs.

5.4.2 Results on the Small Benchmarks

Table 5.1 shows our results on the representative race condition patterns [101, 108, 52]. Columns 1-2 show the name of each benchmark and the number of harmful races (our input). Columns 3-8 show the number of race conditions eliminated by using Type 1, 2, and 3 repairs, respectively. Column 9 shows the total repair time in milliseconds.

Among the 50 known-to-be-harmful race conditions, 6 were eliminated by Type 1 repairs, 28 were eliminated by Type 2 repairs, and 16 were eliminated by Type 3 repairs. The reason why there is a relatively high percentage of Type 1 and 2 repairs is because these benchmarks are created to illustrate common bug patterns; therefore, they are implemented in a relatively simple manner. In particular, almost all of the event handlers are defined statically in these benchmarks, while in real applications, event handlers are often registered at runtime through APIs of third-party libraries such as *jQuery*. Note that the 16 races eliminated by Type 3 repairs cannot be eliminated by any of the Type 1 and 2 repairs, because they involve complex dynamic features such as Ajax, timer, and dynamically added

Table 5.1: Repair results for the small benchmarks.

Name	Races	Type 1	Type 2		Type 3			Repair Time
			2B	2C	3A	3B	3C	(ms)
WebR_ex1	1	0	0	0	1			10
WebR_ex2	1	0	1	0				17
WebR_ex3	1	0	0	1				16
WebR_ex4	1	0	1	0				37
WebR_ex5	1	0	1	0				13
kaist_ex1	1	1	0	0				19
kaist_ex2	1	0	0	0		1		21
kaist_ex3	3	1	0	2				41
EventR_ex1	1	1	0	0				18
EventR_ex2	1	0	0	0		1		10
EventR_tut	3	0	3	0				66
kaist_case1	8	0	0	2	6			192
kaist_case2	3	0	1	1		1		26
kaist_case3	3	1	0	0	2			16
kaist_case4	5	0	0	3	1	1		34
kaist_case5	3	0	1	2				25
kaist_case6	3	2	0	1				114
kaist_case7	7	0	3	3	1			48
kaist_case8	3	0	0	2		1		21
Total	50	6	11	17	11	5	0	744

DOM nodes and event handlers.

5.4.3 Results on the Real Websites

Table 5.2 shows our results for the 122 race conditions from the 42 websites of Fortune-500 companies. Columns 1-2 show the URL of each website and the number of harmful race conditions (our input). Columns 3-5 show the number of race conditions eliminated by our Type 1, Type 2A, and Type 2B repairs, respectively. Columns 6-8 show the number of race conditions eliminated by our Type 3A, 3B, and 3C repairs, respectively. Column 7 shows the repair time in milliseconds.

Recall that we have applied the EventRacer tool to all of the Fortune-500 companies to generate the initial list of race condition warnings. EventRacer found potential race conditions on 251 of the 500 websites. After applying deterministic replay-based classification, we

identified the 122 harmful race conditions from 42 websites. *RCrepair* successfully repaired all of the 122 race conditions. Among them, 7 were eliminated by Type 1 repairs, 57 were eliminated by Type 2 repairs, and 58 were eliminated by Type 3 repairs. Furthermore, the percentage of Type 2B repairs is particularly large because, in these real applications, almost all visual elements are highly interactive in that they are associated with multiple event handlers. For example, it is not uncommon for each link of a web page to be associated with multiple mouse-related event handlers, e.g., to display and to clean the help information when the user moves the cursor over the link.

In addition to showing that our dynamic analysis-based method is effective in repairing harmful race conditions in real applications, the data in Table 5.2 also show that a purely static analysis-based method would not have been as effective. For example, among the 64 race conditions eliminated by our Type 1 and 2 repairs, only 7 of them belong to Type 1 (11%), while the remaining race conditions involve dynamic features and therefore cannot be easily repaired using static analysis. Similarly, among the 58 race conditions eliminated by our Type 3 repairs, only 7 of them belong to Type 3A (12%), while the remaining ones involve dynamic features. We also found that *jQuery* was one of the most popular JavaScript libraries used in real websites—360 of the 500 websites we studied rely on various versions of *jQuery* (among the 42 websites with harmful races, 32 rely on *jQuery*). One main advantage of our method over purely static analysis-based tools such as ARROW [140] is that our method works well with such external JavaScript libraries.

We are not able to experimentally compare with ARROW since the tool is not yet publicly available. We would like to compare with ARROW [140] as soon as it is available. We are thankful to the developers of ARROW, who have been responsive and helpful while we try to understand the details of their techniques.

The correctness of our repair is guaranteed during the classification phase in PreRepair (Algorithm 8), where we enforce both of the two execution orders of the racing events to implicitly verify the feasibility of the desired execution order. After the repair, we invoke

Table 5.2: Repair results for the real websites.

Name	Races	Type 1	Type 2		Type 3			Repair Time
			2A	2B	3A	3B	3C	(ms)
www.newyorklife.com	2			2				331
www.aa.com	2						2	115
honeywell.com	1					1		183
www.cognizant.com	10			10				659
www.loews.com	1						1	53
www.marathonoil.com	1						1	55
www.mcdonalds.com	2				1	1		144
www.regions.com	2	2						91
www.bd.com	2			2				89
www.campbellsoupcompany.com	1						1	253
www.sandisk.com	2			1			1	130
www.weyerhaeuser.com	16			16				216
www.unum.com	1		1					25
www.altria.com	1			1				227
www.autozone.com	1						1	70
www.dollartree.com	2					2		79
www.avoncompany.com	8			4	3	1		53
www.borgwarner.com	4			2	1	1		468
www.cablevision.com	1					1		35
www.conocophillips.com	1					1		198
www.cbre.com	1					1		289
www.colgatepalmolive.com	1					1		100
www.conagrafoods.com	1					1		115
www.domtar.com	1				1			70
www.dickssportinggoods.com	1				1			383
www.entergy.com	11						11	23
www.wrberkley.com	1	1						40
www.thepantry.com	7			7				74
www.rgare.com	3			3				110
www.pnc.com	1						1	272
www.oshkoshcorporation.com	1			1				18
www.oldrepublic.com	12					12		107
www.mosaicco.com	1			1				19
www.merck.com	2			2				328
www.kbr.com	1			1				29
www.insight.com	2					2		116
www.iheartmedia.com	2			2				75
www.huntingtoningalls.com	2	2						6
www.fnf.com	2	2						119
www.firstam.com	5						5	632
www.eogresources.com	2			1		1		85
www.dollargeneral.com	1					1		217
Total	122	7	1	56	7	27	24	6701

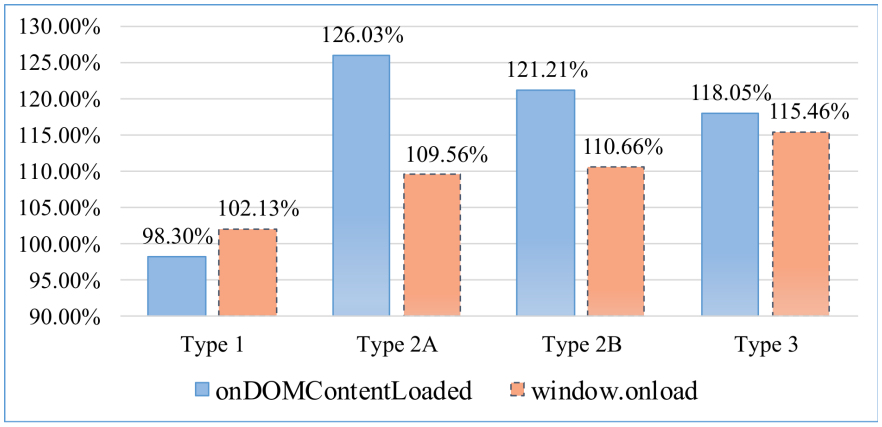


Figure 5.17: Performance overhead on small benchmarks.

PostRepair to confirm, by deterministic replay, that the race condition indeed has been eliminated.

5.4.4 Performance Evaluation

Besides the correctness of the repairs, we also want to know how much impact the repair has on the runtime performance of the application. More specifically, we want to measure the loading time of the web pages, which developers in practice strive to reduce. There are usually two types of measurements of the loading speed (cf. https://developer.mozilla.org/): (1) *DOM ready*, which is associated with *document.onDOMContentLoaded*, fired when the parsing has finished and the DOM has been loaded, but before style sheets, images, and subframes are fully loaded; and (2) *Page ready*, which is associated with *window.onload*, fired when the entire document loading process ends — all objects are in the DOM and all images, scripts, links, and sub-frames are fully loaded.

We evaluated both the *DOM ready* time and the *Page ready* time for each of the repaired web page, and compared it with the original web page. During the evaluation, we have observed that even for the same web page, the loading time may differ significantly; sometimes, one execution may be several times longer than another, because the loading is easily affected

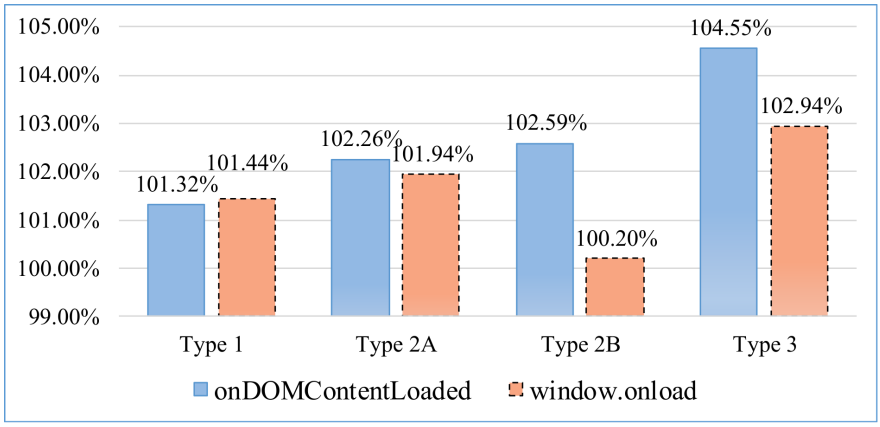


Figure 5.18: Performance overhead on small benchmarks.

by various factors such as caching in the browser, the CPU load, and the network speed. In order to achieve a fair comparison, we executed the original web page and the repaired page 100 times each and computed the average (this process also validated the robustness of our repair in practice, which consistently delivered the intended program behavior).

Figures 5.17 and 5.18 show the performance evaluation results on the two sets of benchmarks, respectively. Respectively, Figures 5.17 shows the results on the small benchmarks, and 5.18 shows the results on the large benchmarks. In each figure, the four columns represent the average time taken to load the repaired web page with respect to the original web page (100% means the loading speed is the same).

In both figures, we have observed almost no performance overhead for the Type 1 repair, which validated our belief that switching the order of elements in the HTML file, when applicable, is the most efficient repair. In contrast, the performance overhead for Type 2 and 3 repairs were slightly higher. However, overall, the overhead remains negligible.

The most important data are the ones from the real websites. Figure 5.18 shows that the average overhead is less than 5% for both DOM ready and Page ready for all types of repairs, indicating that the repairs are efficient enough for practical use. The reason why they have negligible runtime overhead is because, in real web applications, there is often a significantly large number of events than the two events we are trying to control. Furthermore, the racing

```
<div id="hoverMsg1" class="carrot"> Taste it to believe it </div>
<div id="hoverMsg3" class="carrot"> Take our survey </div>
<!-- START OF Google Analytics TAG -->
<script type="text/javascript">
  (function() { var ga = document.createElement('script');
  ga.type = 'text/javascript';
  ga.async = true;
  ga.src = ('https:'== document.location.protocol ? 'https://ssl' :
      'http://www/') + '.google-analytics.com/ga.js';
  var s=document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s); })();
</script>
```

Figure 5.19: Race condition in www.mcdonalds.com.

events often are not computation-intensive events.

5.4.5 Case Studies

www.mcdonalds.com There is a race between two asynchronous JavaScript code blocks that access the same set of elements in the DOM tree. One script is *ga.js* (event ev_a), which is inserted dynamically through an inline JavaScript tag inside the main HTML. The other is *analytics.js* (event ev_b). When ev_a occurs before ev_b, for example, the text of "hoverMsg1" becomes "Explore the menu" after both scripts are loaded, and "hoverMsg3" becomes "Make a difference". However, this is different from the set of original messages (when ev_b occurs before ev_a). Furthermore, there are differences in the *href* attributes of some elements, which can result in difference page being clicked, and differences in the *src* attributes of image tags. This is clearly a bug.

The race condition can be eliminated by enforcing an order between the two racing events. Since both events are from asynchronous scripts, we loaded the original page 100 times and evaluated the content of the messages. We found that most of the time, "hoverMsg1" would show "Taste it to believe it" and "hoverMsg1" would show "Take our survey", and they were consistent with the order of ev_b before ev_a. Furthermore, we found that the average loading

```
<html>
  <body>
    <!-- other Elements -->
   <input id="input_1" type="text" value="Name" onfocus="clearText(this)">
   <input id="input_2" type="text" value="Email" onfocus="clearText(this)">
   <!-- other Elements -->
   <script id="script2" src="nyl-min.js"></script>
   <!-- other Elements -->

  </body>
</html>

<!-- nyl-min.js -->
function clearText(a){ if(a.defaultValue==a.value){a.value=""}}
...
```

Figure 5.20: Race conditions in www.newyorklife.com.

time of ev_b before ev_a was noticeably smaller than ev_a before ev_b. Thus, we conclude that the desired order is ev_b before ev_a. *RCrepair* generated a Type 3 repair, which intercepts the insertion of these two racing events at runtime and ensures the loading of *analytics.js* is before *ga.js*.

We have noticed during our analysis of the experimental results that it is common for real websites to leverage third-party libraries to order the rendering of visual effects. Due to the often large code size, these libraries are usually defined as non-blocking (asynchronous) to allow faster rendering of the main HTML. Therefore, the differences in the script size, the location of the URL, and the network speed can all affect the relative loading order of these libraries, thus leading to unexpected behaviors if they are racing with each other. Although they are difficult to analyze statically, *RCrepair*has been effective in handling such race conditions.

www.newyorklife.com There are two harmful race conditions, the first of which is between *input_1* and the parsing of a synchronous script *nyl-min.js*, and the other is between *input_2* and the same script. They are harmful because *input_1* and *input_2* initially show

the hint values `'Name'` and `'Email'` to the user, but their *onfocus* event handlers, which call *clearText()*, will empty these hint values as soon as the user tries to type into the text areas. This is a standard practice for DOM elements that allowing users to type the information as the input of the web application, such as personal information including name, email, search box, etc. Clearing the initial hint value would offer the user convenience because it saves their time deleting them manually.

Unfortunately, in this example, the script that defines *clearText()* may be parsed after the user typed something into the two fields – assume `'USERTYPED'` is what the user typed. The user typed string and the initial hint string can be interleaved and thus lead to unwanted text contents such as `'NamUSERTYPEDe'` or `'NameUSERTYPED'` instead of just `'USERTYPED'`.

Our solution for this race condition falls into Type 2B, since the shared variable is the callback function itself. We do not apply Type 2A because delaying the registration of the event handler would not help (the same problem would remain). Instead, we temporarily set the two input fields as invisible and disabled, and restore them only after *nyl-min.js* is loaded, so the user cannot interact with them until they are ready. Although the time period in which the elements are responsive is decreased, it is necessary for fixing the bug.

5.5 Related Work

There are several existing tools for detecting race conditions in web applications, some of which are based on static analysis [163] while others are based on dynamic analysis [108, 101, 52]. Static analysis has the advantage of covering all execution paths of the JavaScript code. However, due to the rich set of dynamic features in JavaScript, such as *eval*, asynchronous callback, and dynamic creation of code, JavaScript is difficult to analyze statically [45, 57, 110, 124, 87]. Dynamic analysis tools such as EventRacer [108, 101] and WAVE [52] do not have such limitations. However, dynamic analysis does not have good code coverage [11, 146, 148, 147, 149] and therefore can miss bugs.

Recently, Mutlu et al. [95, 94] proposed a method for detecting race conditions using a combination of dynamic analysis and trace-based static analysis. They first leverage dynamic analysis to obtain an execution trace, and then apply static analysis techniques on the trace, to detect races that may occur in all possible interleavings of the events in that trace. However, their method focuses only on race conditions over persistent storage such as cookie and the server state. Furthermore, none of the existing tools can automatically classify race conditions.

There are also prior work on deterministic replay [89, 9, 18] for web applications, but the goal was to allow developers to revisit any recorded program state to diagnose bugs. A problem related to deterministic replay is the migration of running web applications [15, 81, 80], where the focus is on transferring a session of the running application from one browser to another browser. In this context, they need to address the problem of recording of program states as well. However, the purpose of recording states is significantly different from ours. In particular, none of the prior works on state recording and replay focuses on diagnosing concurrency bugs.

Narayanasamy et al. [96] proposed a replay-based method for automatically classifying data-races in multithreaded C/C++ applications. More specifically, they employed a checkpointing tool to take snapshots of the main memory during the execution of the program, and then compare the snapshots to decide if a data-race is harmful. However, data-races in multithreaded C/C++ programs are significantly different from race conditions in event-driven web applications. Both of the replay and the state comparison require drastically different techniques on these two platforms. Therefore, except for high-level similarity, the technical details of these two approaches are completely different.

Wang et al. [140] developed ARROW, which is a purely static analysis-based tool for automatically patching concurrency bugs in web pages. It models the ordering of events in a web page by statically constructing a causal graph, and then applying an SMT solver-based symbolic analysis to compute the most efficient way of breaking cycles in the causal graph.

Each cycle corresponds to a race condition, and each way of breaking the cycle corresponds to a repair. However, since JavaScript is a highly dynamic language and is known to be difficult to analyze statically, ARROW is inherently limited to the race conditions that can be located by static analysis tools. To handle third-party libraries, the user of ARROW must manually annotate the APIs of these libraries, which is tedious and error prone. Our new method, in contrast, relies on dynamic analysis and therefore does not have these problems.

5.6 Discussions

We have presented *RCrepair*, a dynamic analysis-based tool for automatically diagnosing and repairing race conditions in JavaScript-based web applications. *RCrepair* uses deterministic replay to check whether a race condition is real and harmful, then uses a combination of HTML element reordering and delay with periodic retry to eliminate the erroneous execution order, and finally, uses replay to confirm that the race condition indeed has been fixed. We have implemented the tool and evaluated it on a large set of real web applications. Our experimental results show that the tool is both effective in repairing race conditions and efficient for practical use.

Chapter 6

Conclusions and Future Work

I have proposed a set of new and automated methods for detecting concurrency bugs, avoiding concurrency bugs, diagnosing concurrency bugs and repairing concurrency bugs. They are designed for either shared-memory multithreaded C/C++ programs or event-driven programs such as JavaScript-based client-side web applications.

6.1 Summary

In Chapter 2, I have presented the first runtime analysis-based method for detecting standard and quasi linearizability violations in concurrent data structures. The method works directly on the C/C++ source code and guarantees that all reported violations are real bugs. Furthermore, unlike many of the existing methods, it does not require the user to write functional specifications or manually annotate the linearization points.

In Chapter 3, I have presented the new method for runtime mitigation of concurrency related type-state violations in multithreaded C/C++ applications. The method leverages both static and dynamic analysis techniques to ensure that the erroneous thread interleavings will never occur at run time. Furthermore, it guarantees that the mitigation strategy is always

safe, and if the program has a failure-free interleaving, our new method can always find it.

In Chapter 4, I have presented the new method for automatically classifying race-condition warnings in client-side web applications. It marks a race condition as harmful only if both execution orders of the pair of racing events are indeed feasible and they lead to different program states. The new method builds upon a purely JavaScript-based dynamic analysis framework for implementing the deterministic replay and state comparison, and thus is agnostic to the underlying web browsers and JavaScript engines.

In Chapter 5, I have presented the new method for dynamically analyzing race conditions in web applications and computing potential repairs. The repairs are computed using a combination of HTML file rewriting, which adds the must-happen-before constraints statically, and injection of JavaScript code, which use temporary delay together with periodic retry to eliminate erroneous execution orders. Compared to static analysis based techniques, our new method has the advantage of being more generally applicable, e.g., it can more robustly handle race conditions that involve external JavaScript libraries.

6.2 Future Work

Although the new methods proposed in this dissertation are shown to be both effective and practically applicable, they seem to have raised more questions than they have answered. At this moment, it is clear that a lot more work is needed before we can provide adequate tools to help ordinary programmers write concurrent software correctly as well as efficiently. Specifically, several lines of research have arisen from this dissertation, and they should be pursued in the future.

First, testing and verifying the implementation of concurrent data structures remain difficult tasks in practice. Although our new method is effective in detecting standard and quasi linearizability violations, its performance and scalability still need improvement. Since concurrent data structures are crucial for many applications to leverage the computing power of

multi-core processors, it is important that we keep improving the efficiency and scalability of these testing and verification tools.

Second, high-level concurrency bugs such as type-state violations are common in practice but less well-studied by the research community. Although our new method for mitigating this type of violations at run time is effective, it merely avoids the erroneous thread interleavings that manifest these bugs, without eliminating them from the program. Future work includes the development of new methods for statically repairing these bugs in the program, e.g., by adding new synchronization operations to the source code of the program.

Third, due to the pervasive use of web and mobile applications, debugging event-driven programs are becoming increasingly important. In this context, our purely JavaScript-based deterministic replay framework has been extremely useful, e.g., to help classify harmful race conditions. I believe the new and platform-agnostic dynamic analysis framework can benefit many other applications. For example, it can be used to monitor the exchange of information between the client side and the server. It can also be used to perform systematic exploration of all possible interleavings of the web application.

Finally, automated program repair has become an important research topic. The new method proposed in this dissertation focuses primarily on using dynamic analysis techniques for repairing race conditions in web applications. It would be interesting to see if such technique can be combined with static program analysis to improve the quality of the repair. For example, by statically analyzing the control and data flows of the program, we may be able to compute repairs that have lower runtime overhead. Another research direction is to automatically repair concurrency bugs in server-side web applications.

Bibliography

[1] K. Adhikari, J. Street, C. Wang, Y. Liu, and S. Zhang. Verifying a quantitative relaxation of linearizability via refinement. In *International SPIN Symposium on Model Checking of Software*, 2013.

[2] K. Adhikari, J. Street, C. Wang, Y. Liu, and S. Zhang. Verifying a quantitative relaxation of linearizability via refinement. *International Journal on Software Tools for Technology Transfer*, 2015.

[3] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVM: A low-level virtual instruction set architecture. In *ACM/IEEE international symposium on Microarchitecture*, San Diego, California, Dec 2003.

[4] Y. Afek, G. Korland, and E. Yanovsky. Quasi-Linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410, 2010.

[5] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, 54(5):3, 2010.

[6] G. Agosta, A. Barenghi, and G. Pelosi. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the Design Automation Conference*, pages 77–82, 2012.

[7] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding javascript event-based interactions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 367–377, 2014.

[8] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[9] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 403–410, 2011.

[10] M. Arnold, M. T. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, 21(1):2, 2011.

[11] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 571–580, 2011.

[12] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[13] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 589–608, 2007.

[14] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[15] F. Bellucci, G. Ghiani, F. Paternò, and C. Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd*

ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011, pages 105–110, 2011.

[16] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: Runtime enforcement for reactive systems. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 533–548, 2015.

[17] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 330–340, 2010.

[18] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, pages 473–484, 2013.

[19] P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *International Conference on Computer Aided Verification*, pages 465–479, 2010.

[20] F. Chen and G. Rosu. Parametric and sliced causality. In *International Conference on Computer Aided Verification*, pages 240–253. Springer, 2007.

[21] S. Christey(editor). Top 25 most dangerous programming errors. *CWE/SANS report*, 2009.

[22] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV'00)*, pages 154–169. Springer, 2000. LNCS 1855.

[23] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[24] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490, 2004.

[25] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[26] J. V. Deshmukh, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Logical concurrency control from sequential proofs. In *European Symposium on Programming*, pages 226–245, 2010.

[27] H. Eldib and C. Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *International Conference on Formal Methods in Computer-Aided Design*, 2013.

[28] H. Eldib and C. Wang. An SMT based method for optimizing arithmetic computations in embedded software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(11):1611–1622, 2014.

[29] H. Eldib and C. Wang. Synthesis of masking countermeasures against side channel attacks. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 114–130, 2014.

[30] H. Eldib, C. Wang, and P. Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[31] H. Eldib, C. Wang, and P. Schaumont. SMT based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2014.

[32] H. Eldib, C. Wang, M. Taha, and P. Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *Proceedings of the Design Automation Conference*, pages 209:1–209:6, 2014.

[33] H. Eldib, C. Wang, M. Taha, and P. Schaumont. Quantitative Masking Strength: Quantifying the side-channel resistance of masked software code. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2015.

[34] H. Eldib, M. Wu, and C. Wang. Synthesis of fault-attack countermeasures for cryptographic circuits. In *International Conference on Computer Aided Verification*, 2016.

[35] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[36] A. Farzan and P. Madhusudan. Causal atomicity. In *International Conference on Computer Aided Verification*, pages 315–328, 2006.

[37] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *International Conference on Computer Aided Verification*, pages 52–65, 2008.

[38] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Parallel and Distributed Processing Symposium*, 2004.

[39] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[40] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[41] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[42] M. Gallaher and B. Kropp. Economic impacts of inadequate infrastructure for software testing. Technical Report RTI Report for National Institute of Standards and Technology (NIST), Health, Social, and Economics Research, RTI, Research Triangle Park, NC 27709, 2002.

[43] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 459–470, 2015.

[44] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250, 2011.

[45] S. Guarnieri and V. B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168, 2009.

[46] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[47] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Conf. Computing Frontiers*, page 17, 2013.

[48] T. A. Henzinger, A. Sezgin, C. M. Kirsch, H. Payer, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2013.

[49] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[50] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[51] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[52] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side java script web applications. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 61–70, 2014.

[53] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[54] J. Huang, J. Zhou, and C. Zhang. Scaling predictive analysis of concurrent programs by removing trace redundancy. *ACM Trans. Softw. Eng. Methodol.*, 22(1):8, 2013.

[55] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 57–73, 2015.

[56] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev. Stateless model checking of event-driven applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 57–73, 2015.

[57] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[58] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, 2011.

[59] G. Jin, W. Zhang, and D. Deng. Automated concurrency-bug fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 221–236, 2012.

[60] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 288–296, 2008.

[61] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 295–308, 2008.

[62] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Static analysis for concurrent programs with applications to data race detection. *International Journal on Software Tools for Technology Transfer*, 15(4):321–336, 2013.

[63] V. Kahlon and C. Wang. Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In *International Conference on Computer Aided Verification*, pages 434–449, 2010.

[64] V. Kahlon and C. Wang. Lock removal for concurrent trace programs. In *International Conference on Computer Aided Verification*, pages 227–242, 2012.

[65] E. Kandrot and B. Eich. Our JavaScript is 3x slower than IE's. *The Mozila Project Website*, Sept. 2000.

[66] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 185–198, 2012.

[67] S. Khoshnood, M. Kusano, and C. Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, 2015.

[68] C. M. Kirsch and H. Payer. Incorrect systems: it's not the problem, it's the solution. In *Proceedings of the Design Automation Conference*, pages 913–917, 2012.

[69] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent fifo queues. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 273–287, 2012.

[70] B. Krebs. A time to path II: Mozilla. *The Washington Post Security Fix Blog*, Feb. 2006.

[71] M. Kuperstein, M. T. Vechev, and E. Yahav. Automatic inference of memory fences. In *International Conference on Formal Methods in Computer-Aided Design*, pages 111–119, 2010.

[72] M. Kusano, A. Chattopadhyay, and C. Wang. Dynamic invariant generation for concurrent programs. In *International Conference on Software Engineering*, 2015.

[73] M. Kusano and C. Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 722–725, 2013.

[74] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[75] M. Kusano and C. Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[76] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[77] Z. Letko, T. Vojnar, and B. Krena. AtomRace: data race and atomicity violation detector and healer. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, page 7. ACM, 2008.

[78] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, pages 299–309, 2012.

[79] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 321–337, Berlin, Heidelberg, 2009. Springer-Verlag.

[80] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Imagen: runtime migration of browser sessions for javascript web applications. In *International World Wide Web Conference*, pages 815–826, 2013.

[81] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Live migration of javascript web apps. In *International World Wide Web Conference*, pages 241–244, 2013.

[82] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[83] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[84] W. Lu, Y. Yang, L. Wang, W. Xing, and X. Che. A novel concurrent generalized deadlock detection algorithm in distributed systems. In *Algorithms and Architectures*

for Parallel Processing - 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part II, pages 479–493, 2015.

[85] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-Aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1):73–83, 2009.

[86] Q. Luo and G. Rosu. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *International Symposium on Software Testing and Analysis*, pages 156–166, 2013.

[87] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 499–509, 2013.

[88] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.

[89] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 159–174, 2010.

[90] Microsoft. Revamping the Microsoft security bulletin release process. *Microsoft TechNet Security Bulletins*, Feb. 2005.

[91] A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. pages 58–75, 2012.

[92] M. Musuvathi and S. Qadeer. CHESS: Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation*, pages 15–16. Springer, 2006.

[93] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[94] E. Mutlu, S. Tasiran, and B. Livshits. I know it when I see it: Observable races in javascript applications. In *Proceedings of the Workshop on Dynamic Languages and Applications, Dyla 2014, Edinburgh, United Kingdom, June 9-11, 2014*, pages 1:1–1:7, 2014.

[95] E. Mutlu, S. Tasiran, and B. Livshits. Detecting javascript races that matter. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 381–392, 2015.

[96] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 22–31, 2007.

[97] N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184, 2016.

[98] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[99] H. Payer, H. Röck, C. M. Kirsch, and A. Sokolova. Scalability versus semantics of concurrent fifo queues. In *ACM Symposium on Principles of Distributed Computing*, pages 331–332, 2011.

[100] R. Pegoraro. Apple updates Leopard – again. *The Washington Post*, Feb. 2008.

[101] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 251–262, 2012.

[102] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[103] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by data race detection. *J. Log. Algebr. Meth. Program.*, 83(5-6):400–426, 2014.

[104] S. K. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: dynamically ensuring isolation in comcurrent programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, 2009.

[105] R. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[106] R. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, pages 81–98, 1989.

[107] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. G. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–184, 2009.

[108] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 151–166, 2013.

[109] V. Raychev, M. T. Vechev, and E. Yahav. Automatic synthesis of deterministic concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 283–303, 2013.

[110] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In *The European Conference on Object-Oriented Programming*, pages 52–78, 2011.

[111] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

[112] U. Salzburg. Scal: High-performance multicore-scalable data structures. URL: http://scal.cs.uni-salzburg.at/.

[113] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[114] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[115] T.-F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*, pages 136–150, 2012.

[116] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[117] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007. USENIX Association.

[118] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, 2011.

[119] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *Formal Methods and Models for Codesign*, pages 99–108, 2011.

[120] N. Sinha and C. Wang. Staged concurrent program analysis. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 47–56, 2010.

[121] N. Sinha and C. Wang. On interference abstractions. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 423–434, 2011.

[122] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[123] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[124] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *The European Conference on Object-Oriented Programming*, pages 435–458, 2012.

[125] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[126] C. Sung, M. Kusano, and C. Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[127] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, Berlin, Heidelberg, 2009. Springer-Verlag.

[128] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[129] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *International SPIN Workshop on Model Checking Software*, pages 261–278, 2009.

[130] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 139–154, 2009.

[131] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[132] C. Wang and M. Ganai. Predicting concurrency failures in generalized traces of x86 executables. In *International Conference on Runtime Verification*, pages 4–18, Sept. 2011.

[133] C. Wang and K. Hoang. Precisely deciding control state reachability in concurrent traces with limited observability. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 376–394, 2014.

[134] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272, 2009.

[135] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 328–342, 2010.

[136] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[137] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[138] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[139] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[140] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster. ARROW: Automated repair of races on client-side web pages. In *International Symposium on Software Testing and Analysis*, 2016.

[141] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[142] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 252–263, 2009.

[143] Y. Wang, P. Liu, T. Kelly, S. Lafortune, S. A. Reveliotis, and C. Zhang. On atomicity enforcement in concurrent software via discrete event systems theory. In *IEEE Conference on Decision and Control*, pages 7230–7237, 2012.

[144] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 295–306, 2009.

[145] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 19–34, 2011.

[146] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *International Symposium on Software Testing and Analysis*, pages 336–346, 2013.

[147] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 1–26, 2014.

[148] S. Wei and B. G. Ryder. Taming the dynamic behavior of javascript. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '14, Portland, OR, USA, October 20-24, 2014 - Companion Volume*, pages 61–62, 2014.

[149] S. Wei and B. G. Ryder. Adaptive context-sensitive analysis for javascript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 712–734, 2015.

[150] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 205–216, 2012.

[151] M. Wu, H. Zeng, and C. Wang. Synthesizing runtime enforcer of safety properties under burst error. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 65–81, 2016.

[152] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.

[153] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah, 2008.

[154] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN Workshop on Model Checking Software*, pages 288–305, 2008.

[155] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN workshop on Model Checking Software*, pages 279–295, 2009.

[156] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *International Symposium on Software Testing and Analysis*, pages 169–179, 2002.

[157] F. Yu, C. Wang, A. Gupta, and T. Bultan. Modular verification of web services using efficient symbolic encoding and summarization. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 192–202, 2008.

[158] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, pages 325–336, 2009.

[159] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[160] L. Zhang, A. Chattopadhyay, and C. Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 4–14, 2013.

[161] L. Zhang, A. Chattopadhyay, and C. Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *IEEE Transactions on Software Engineering*, 2015.

[162] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.

[163] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *International World Wide Web Conference*, pages 805–814, 2011.

[164] H. Zhu, G. Petri, and S. Jagannathan. Poling: SMT aided linearizability proofs. In *International Conference on Computer Aided Verification*, pages 3–19, 2015.