Runtime Prevention of Concurrency Related Type-State Violations in Multithreaded Applications

Lu Zhang
Department of ECE, Virginia Tech
Blacksburg, VA 24061, USA
zhanglu@vt.edu

Chao Wang
Department of ECE, Virginia Tech
Blacksburg, VA 24061, USA
chaowang@vt.edu

ABSTRACT

We 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. Our 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. We have implemented the new method in a tool based on the LLVM compiler framework. Our 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.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability

Keywords

Type state automaton, race condition, failure mitigation, program repair, model checking, partial order reduction

1. INTRODUCTION

We propose a unified runtime mitigation framework for suppressing *concurrency* related type-state violations in multithreaded C/C++ applications. Bugs in multithreaded applications are notorious difficult to detect and fix due to 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 [29, 43, 20, 17, 40, 13, 18, 38], 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, focus on method calls of the shared objects.

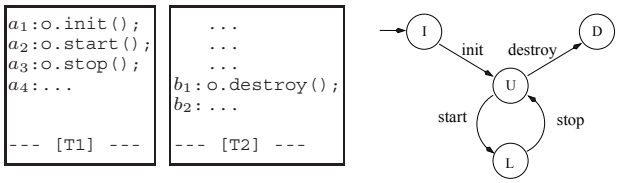


Figure 1: A client program and the type-state automaton. For ease of drawing, we have omitted the non-accepting state. However, there is an implicit edge from every state to the non-accepting state. For example, from state I, executing any method other than `init()` leads to the non-accepting state.

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 defines the application programming interface (API). Type-state automaton can be used to specify the temporal aspects of the API that standard type systems cannot express. As illustrated in Figure 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 1 (left), its method call sequence depends on the thread execution order. This may lead to concurrency bugs. For example, the automaton in Figure 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'14, July 21-25, 2014, San Jose, CA, USA.
Copyright 2014 ACM 978-1-4503-2645-2/14/07 ...$15.00.

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 program so that it emits only correct method call sequences.

The overall flow of our method is illustrated in Figure 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.

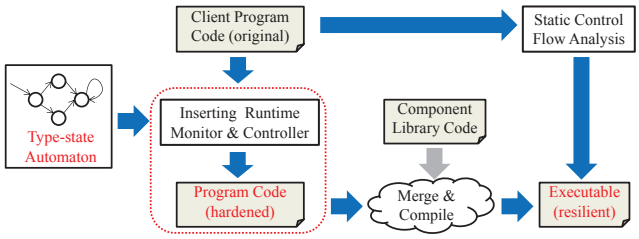


Figure 2: Preventing concurrency type-state errors.

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 paper 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 [21] or bailed out by adding a timeout.

Our method also differs from most of the existing methods for tolerating concurrency bugs [29, 43, 20, 17, 40, 13, 18, 38], 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 [19] 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%.

2. MOTIVATING EXAMPLE

We provide an overview of the new runtime analysis and mitigation method by using the example in Figure 1. Recall that the buggy program involves at least the two threads in Figure 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 [12]. Figure 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. According to the

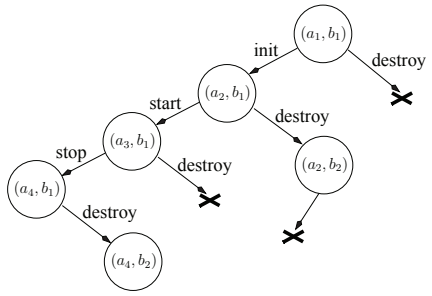


Figure 3: The combined state transition system for the client program M and the automaton A in Figure 1. Each cross represents a bad automaton state.

```
a1:pre_call(&o,'init');
   o.init();
a2:pre_call(&o,'start');
   o.start();
a3:pre_call(&o,'stop');
   o.stop();
a4:...

--- [T1] ---
```

```
   ...
   ...
   ...
   ...
   ...
b1:pre_call(&o,'destroy');
   o.destroy();
b2:
--- [T2] ---
```

Figure 4: An instrumented client program with `pre_call()` controlling the execution of method calls.

type-state automaton in Figure 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 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). 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 [3, 12]. 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. THE SCOPE AND CHALLENGES

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* [31]. 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 [31] – this is a well known theoretical limitation of type-state automaton.

Our paper is not about lifting this theoretical limitation. Instead, our paper 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 1 may give the impression that runtime prevention of concurrency related type-state errors is easy. However, this is not the case. Consider Figure 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 immedi-

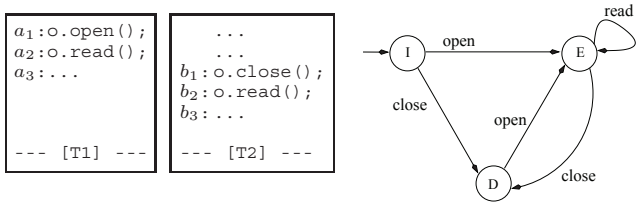


Figure 5: A client program and the type-state automaton. All valid thread interleavings must start with $b_1, a_2, \ldots$

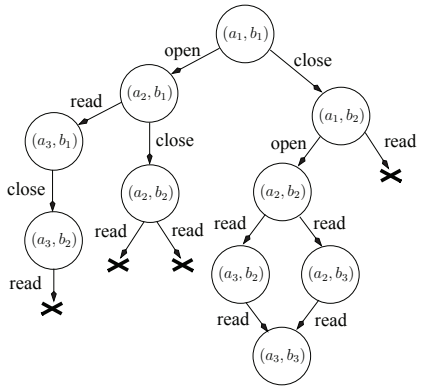


Figure 6: State transitions for the example in Figure 5.

ately 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 paper – 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 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.

4. PRELIMINARIES

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

4.1 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 [31], 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., [39]) 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. [5]), which make the automaton even more expressive. However, we have decided not to explore these extensions due to performance concerns.

4.2 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 [3, 12]. 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*).

4.3 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.

5. 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.

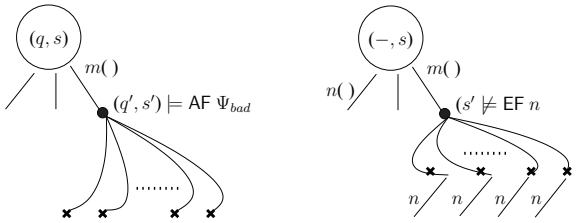
5.1 The Theory

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 which 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 [3, 12]. 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 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.



(a) theoretical method (b) practical method

Figure 7: Illustrating the correctness proof of our runtime analysis and error mitigation strategy.

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 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 1. 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.

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 [11]. During our experiments, the delay time starts from MIN=1ms to MAX=1s.

Algorithm 1 (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: }
```

EXAMPLE 1. *Consider the client program in Figure 1 and the state transition graph in Figure 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.*

EXAMPLE 2. *Consider the client program in Figure 5 and the state transition graph in Figure 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 6.*

To determine whether a running thread is *enabled*, which is required by Line 3 of Algorithm 1, 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 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.

5.2 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 1 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 1: If thread T is the only enabled thread at that time, 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 1 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 [3, 12]. 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 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 4, 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.

6. THE PRACTICAL 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.

6.1 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 1.

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 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 2. 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 2 (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 8.

6.2 Rules Based on Limited Observability of the 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 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 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 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 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 will 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 2 in the same way as we use Rule 2, except that the patterns must to be collected *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 7.

7. STATIC CONTROL FLOW ANALYSIS

The static analysis as required in Figure 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.

7.1 Collecting Future Events of a Given Event

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 → futureMethods)`.

These tables are computed statically *a priori* and then made available at runtime. Our runtime mitigation algorithm will leverage them for evaluating Rule 3 in Algorithm 2, 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.

7.2 Improving the Runtime Performance

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 7.1 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 2. This approach is good for performance, but may reduce the effectiveness in suppressing bugs. For example, in Figure 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.

8. 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 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 [43]. `Transmission-1.42` is a multi-thread 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.

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 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 occurred and the run 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 run time. These experiments were conducted on a workstation with 2.7 GHz Intel Core i7 (with four logical processors) and 8GB memory.

Table 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 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.

Table 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

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 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 8 shows the results. The x-axis are the the benchmarks. The y-axis are the runtime overhead in percentage. The results show that our prevention method has only a small performance overhead. Except for `THRIFT`, all the other benchmarks have a runtime overhead of less than 5%. The reason why `THRIFT` has a larger runtime overhead is because the method calls of the shared object are inside a computation-intensive loop, whereas in all the other examples, the shared objects (such as `timer` and `file_reader`) are not insider computation-intensive loops. On average, our results show a runtime performance overhead of 1.45%.

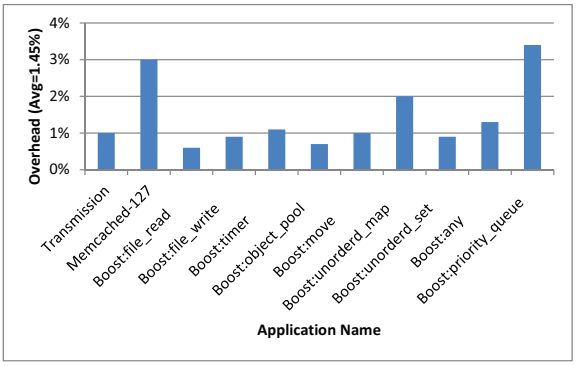


Figure 8: The overhead of runtime failure mitigation.

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 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 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

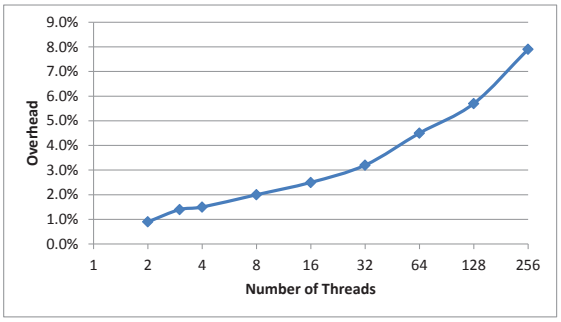


Figure 9: Runtime overhead on `Boost:file_write`: while the x-axis is in logarithmic scale, the y-axis is in linear scale.

overhead (on average 1.45%). To put it into context, in the AVIO-S [19] 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.

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 white-list 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.

9. RELATED WORK

Type-state automaton was first introduced by Strom and Yemini [31]. Since then, many static (e.g., [4, 5, 9]) and dynamic (e.g., [1, 2]) 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 [14] and 2ndStrike [10]. 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 prevention 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 [43] 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 [20, 17, 40, 13] and data-races [26, 29]. Wu et al. [41] use a set of user-specified order constraints to hot-patch a running concurrent application, by selectively inserting lock statements to the code. Dimmunix [15] and Gadara [36, 37] are tools for avoiding previously encountered deadlocks. Recent work by Liu *et al.* [18, 38] has extended the idea to avoid atomicity violations. 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 [21], 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 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 control 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 [23, 24, 35, 30, 33, 44, 34] 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.

Our method is also related to the line of theoretical work on program synthesis [8, 22, 25, 32] and controller synthesis [27, 28]. In this context, Vechev *et al.* [32] infer high-level synchronizations [16] from safety specifications. Deshmukh *et al.* [7] require pre- and post-conditions. Deng *et al.* [6] and Yavuz-Kahveci *et al.* [42] semi-automatically map high-level models into low-level implementations. However, these methods are all offline methods for generating or patching the program code; This is a completely different problem 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 micro seconds.

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.

10. CONCLUSION

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.

11. ACKNOWLEDGMENT

The work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

12. REFERENCES

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

[2] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 589–608, 2007.

[3] 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.

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

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

[6] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[7] Jyotirmoy V. Deshmukh, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In *European Symposium on Programming*, pages 226–245, 2010.

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

[9] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[10] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250, 2011.

[11] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

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

[13] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, 2011.

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

[15] Horatiu Jula, Daniel M. Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 295–308, 2008.

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

[17] Zdenek Letko, Tomás Vojnar, and Bohuslav 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.

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

[19] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[20] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. Atom-Aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1):73–83, 2009.

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

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

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

[24] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

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

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

[27] 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.

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

[29] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin G. Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–184, 2009.

[30] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

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

[32] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 139–154, 2009.

[33] Chao Wang and Malay Ganai. Predicting concurrency failures in generalized traces of x86 executables. In *International Conference on Runtime Verification*, pages 4–18, September 2011.

[34] Chao Wang and Kevin 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.

[35] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[36] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[37] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 252–263, 2009.

[38] Yin Wang, Peng Liu, Terence Kelly, Stéphane Lafortune, Spyros A. Reveliotis, and Charles Zhang. On atomicity enforcement in concurrent software via discrete event systems theory. In *IEEE Conference on Decision and Control*, pages 7230–7237, 2012.

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

[40] Dasarath Weeratunge, Xiangyu Zhang, and Suresh 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.

[41] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng 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.

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

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

[44] Lu Zhang, Arijit Chattopadhyay, and Chao Wang. Round-Up: Runtime checking quasi linearizability of concurrent data structures. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 4–14, 2013.