Automatic Empirical Failure Avoidance for Concurrent Software

Brandon Lucia Luis Ceze

University of Washington, Department of Computer Science and Engineering
{blucia0a,luisceze}@cs.washington.edu
http://sampa.cs.washington.edu

Abstract

We propose Aviso, a system for avoiding failures due to concurrency errors. Aviso monitors program events during multithreaded execution. When a failure occurs, Aviso represents sequences of events from the failing execution as state-machines that we call SHIELDs. Aviso can use SHIELDs to avoid event sequences that lead to failures. After a failure, Aviso creates many candidate SHIELDs representing possible causes. Aviso then empirically vets them to determine which best avoid failures. We implemented Aviso and showed it increases reliability, and imposes low performance overhead (0-30%).

1. Introduction

Bugs in concurrent programs are hard to find and fix. They arise due to thread interactions via synchronization and shared data that were unforeseen during development. Among common concurrency errors are atomicity violations, in which one thread's access to shared state is incorrectly permitted to interleave between a pair of accesses in another thread. Ordering violations are another type of error, where different threads' operations occur in an order leading to a failure. Most prior work has focused on *detecting* concurrency errors, which typically involves identifying suspect patterns of operations specific to certain bug types. Unfortunately, even with debugging tools, concurrency bugs end up in deployed code, leading to costly failures.

Recent research has explored *avoidance* of concurrency-related failures. The premise of most such work is to determine sequences of operations in different threads that are likely to lead to failures and prevent those sequences. This general idea has seen several incarnations, some of which avoid bad sequences with special hardware [8, 9, 17, 18], some using memory protections [14, 15], and others with programmer annotations that explicitly prohibit behavior [16]. Fixing concurrency errors is hard, often takes a long time [7], and bugs are often "fixed" incorrectly. Avoidance systems prevent failures in deployed code, decreasing the cost in reliability while a fix can ultimately be written and applied.

In this paper we propose Aviso, a novel fully automated system for bug avoidance. The core idea behind Aviso is to monitor program execution and automatically learn which sequences of events lead to failures. Aviso can then avoid these sequences in future executions to avoid failures. Aviso monitors synchronization operations and data accesses and watches for failures. Based on a history of events preceding a failure, Aviso generates a set of interleaving constraints that may prevent the failure, and uses an empirical process to determine which most effectively avoids the failure. In summary, this paper makes the following contributions: **(1)**A fully automatic technique for concurrency error avoidance. **(2)** Several static and dynamic techniques to efficiently find and collect interesting events from executions. **(3)** A new state machine abstraction for concurrency error behavior. **(4)** Aviso: a framework and runtime that generates state machines and uses them to prevent failures in a single software instance or cooperatively in a replicated system. **(5)** An evaluation of Aviso using several real-world applications, demonstrating its efficacy and efficiency.

The rest of this paper is organized as follows. Section 2 provides an overview of Aviso. Section 3 discusses Aviso's approach to event monitoring. Section 4 discusses how Aviso generates candidate SHIELDs after a failure. Section 5 describes how Aviso determines which candidate SHIELDs prevent failures, and how SHIELDs can be shared between systems. Section 6 provides implementation details and evaluation. Section 7 contrasts Aviso with prior work, and Section 8 concludes.

2. Aviso

There are several key aspects to Aviso: efficiently monitoring program events; automatically determining sequences of events responsible for a failure; generating schedule constraints likely to avoid failures, and determining which are most effective; and using schedule constraints to prevent failures. This section provides an overview of Aviso and its design requirements.

2.1 Overview

Aviso monitors program events during execution and keeps a short history of events. Aviso also monitors for failures. When a failure occurs, Aviso determines which event sequence in the history was most likely the cause, and tries to prevent that sequence in the future. Aviso abstracts event sequences as finite state machines. After a failure, Aviso examines the event history and generates a state machine for each interesting event sequence it contains. In subsequent executions, Aviso runs each state machine to determine when the program is about to execute the sequence of events it represents. When such a situation arises, Aviso delays some events in the sequence, perturbing the execution. If the state machine corresponds to the sequence that caused the failure, the delay avoids the sequence, avoiding the failure.

Aviso generates a large set of state machines from its history after a failing run. Each state machine is a *hypothesis* about how to prevent the failure. Aviso empirically evaluates each hypothesis, to determine which ones actually prevent the failure. We call effective hypotheses **S**tate machines for **H**iding **I**nteractions **E**nding in **L**ikely **D**efects, or **SHIELDs**. To leverage large collections of computers running the same program, a SHIELD that consistently prevents a failure on one machine is distributed to other machines running the same program, to share its avoidance capability.

2.2 Building the SHIELD for a Failure

The key concept in Aviso is the mapping between program errors and SHIELDs. As with usual state machines, a SHIELD is made up of *states* and *transitions*. A SHIELD abstractly represents a sequence of events. Each state in a SHIELD represents a point in the progression through the sequence. Transitions between states are triggered when events in the sequence execute. Events can be arbitrary program operations but we focus on events that are likely to be related to concurrency errors, such as synchronization operations, asynchronous signal handlers and memory accesses to shared data. We define events in more detail in Section 3. *Failure states* are states in a SHIELD

corresponding to points in the interleaving sequence where buggy behavior manifests.

Example: AGet Atomicity Violation Bug Figure 1 illustrates the mapping between concurrency errors and SHIELDs using an example from AGet-0.4, a multithreaded download tool. Figure 1(a) shows a buggy execution. The error is an atomicity violation: Thread 1 writes to a file (line 116), then adds the number of bytes written to `bwritten` (line 121). AGet fails when Thread 2 reads the stale value of `bwritten` at line 41, between thread 1's file write and counter update. Note that if Thread 2 is delayed, and Thread 1 finishes both operations, the failure is avoided.

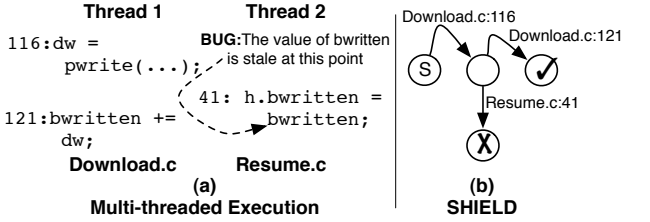

Figure 1. Atomicity bug in AGet-0.4. (a) shows the code involved in the bug. (b) shows an SHIELD representing the interleaving.

Figure 1(b) shows a SHIELD for the bug in 1(a). The start state is labeled "S". The failure state is labeled "×". The "✓" state is the only non-failure state reachable from the center state. When the SHIELD reaches × or ✓, it terminates. From the start state, the SHIELD transitions to the center state after the `pwrite(...)` executes. From this state there are two possibilities: if Thread 2 executes its read, the SHIELD transitions to the × state, indicating failure. If Thread 1 executes its increment instead, the SHIELD transitions to the ✓ state and terminates. We now explain how Aviso manipulates program execution to avoid failure states.

2.3 Bug Avoidance with SHIELDs

Aviso maintains a set of SHIELDs, each corresponding to an interleaving of events that leads to a failure. It monitor events as a program executes, and updates the current state of the SHIELDs when necessary. When a SHIELD enters a state with an outgoing transition to a failure state (like the center state in Figure 1), Aviso prevents the execution of events that lead to that failure state by delaying their execution. While these failure triggering events are prohibited from executing, other threads can execute other operations, leading the SHIELD (and equivalently, the execution) back to a safe state (like the ✓ state). The delayed event is eventually allowed to proceed, but because the sequence of events executed has changed, it will not cause the transition to a failure state. By preventing SHIELDs from transitioning to failure states, sequences that lead to error behavior are prohibited.

2.4 System Design

Figure 2 shows a block-diagram of Aviso's components, and lists the responsibilities of each component. There are four important components: (1) The programmer, (2) The compiler and profiler, (3) The runtime system, and (4) The framework. The programmer's responsibilities are mostly unchanged. At development time, Aviso's profiler and compiler component determines what program events should be monitored, and adds events to the program's binary. The Aviso runtime is linked to the event-aware binary. It monitors events, and watches for failures. The Aviso runtime also takes avoidance actions on some events to prevent failures from occurring. Aviso-enabled programs run in the Aviso framework. The framework uses information collected from a failing run by the runtime to identify a SHIELD that can avoid the failure. The framework then distributes failure-avoiding SHIELDs to other systems to share the failure avoidance capability.

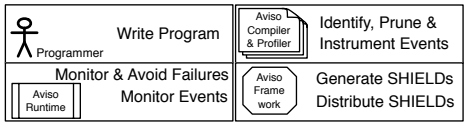

Figure 2. An overview of the responsibilities of each of Aviso's components. The programmer's responsibilities are unchanged.

3. Monitoring Events and Failures

Collecting events relevant to errors efficiently is crucial. Aviso uses several analyses to identify which events in a program to monitor.

3.1 Identifying Relevant Program Events

Our goal is to be general enough to prevent a broad class of errors in deployment. This goal presents conflicting design constraints: For generality, we should monitor many types of events to capture a large variety of failures. However, monitoring imposes overhead, and use in deployment necessitates high performance.

Aviso focuses on concurrency errors, so we monitor synchronization events, signal events, and sharing events. Synchronization events are all lock and unlock operations, as well as thread spawn and join operations. These events can be identified by matching synchronization library (*e.g.*, `pthread`) calls. Aviso can be set up to handle custom synchronization (*e.g.*, CAS) at configuration time.

Signal events are functions that handle signals. They are of interest because signals may be delivered and handled asynchronously. Signal events can be identified by instrumenting signal handler registration functions (*e.g.*, `signal()`), and functions that explicitly wait for signal delivery (*e.g.*, `sigwait()`). Sharing Events are more difficult to identify because in general sharing events cannot be identified by looking only at syntactic properties of a program. Instead, Aviso identifies sharing events using a *sharing profiler* before the deployment of the application – *i.e.*, during testing. The sharing profiler monitors threads' accesses to shared data. When a thread accesses data, if the data has been accessed by another thread during the execution, the operation the thread is executing is a sharing event. Sharing events are reported by the profiler and compiled into the deployment binary by Aviso's instrumenting compiler. Aviso identifies events with the instruction address and call stack at the point when the event occurs.

3.2 Pruning and Instrumenting Events

Performance is essential to Aviso. If the runtime has to handle events too frequently, performance overheads will be unacceptably high. To mitigate the overhead, we use three techniques to reduce the number of events. First, our instrumenting compiler does *dominance pruning* to eliminate redundant events. Second, using profiled information we perform *co-occurrence pruning* to coalesce redundant events. Finally, we use dynamic analysis to do *online pruning*.

Dominance Pruning Analysis Aviso's compiler computes each instruction's dominators, to use them to prune redundant events. For a pair of events (p, q) if p dominates q, then for every execution of q, there is a prior execution of p. Hence, tracking both p and q is redundant. In this situation, we remove q from the set of candidate events. Note, if p and q are far apart, pruning may discard useful events. However, our analysis does not pose problems for two reasons. First, it operates at function scope, bounding the distance between p and q to the length of a function. Second, if events are far apart, dominance still conveys information about the interleaving of events on involved control flow paths – less precise, but still useful for preventing failures.

Co-Occurrence Pruning Aviso prunes events by finding pairs of events that occur within a short interval (100μs) of one another during profiling, and coalescing the pair into a single event. Pairs are coalesced if the expected result of doing so is a significant reduction in *dynamic* events in future executions. We omit a full formal description of co-occurrence pruning due to space constraints.

Online Pruning Aviso uses *online pruning* to adaptively reduce the number of events handled. During execution, Aviso tracks the interval between consecutive events and uses an adaptive analysis to change how an event is handled based on the interval between events. The intuition behind the analysis is that if two events occur in a short interval, they are likely redundant. There are three cases: If the interval is longer than 200μs, Aviso processes the event normally. If the interval is between 10μs and 200μs Aviso only records a *truncated backtrace*, containing only one return address. Finally, if the interval is less than 10μs, Aviso discards the event. Discarding events is, in effect, dynamically coalescing sequences of events into a single event.

3.3 Tracing Important Program Events

To generate SHIELDs, Aviso uses events from a single failing execution. Aviso focuses on program events in a failing run that occurred just before the failure. These events are likely to be related to the failure, because *some* code point must trigger the failure (*e.g.*, cause a crash, emit buggy output, *etc.*); this event occurs shortly before the buggy behavior manifests. Hence, a backward scan over an event trace from failure is likely to encounter events involved in the cause.

Aviso maintains a history of recently executed events, called the *Recent Past Buffer*, or RPB. The RPB is a fixed-size queue storing on the order of hundreds of events (we used 1000 events). When an event executes, the oldest event in the RPB is dequeued and discarded, and the newest event is enqueued. When a failure occurs, the RPB contains a history of the execution's final moments, likely including the events that led to the failure.

3.4 Monitoring Program Failures

Aviso examines the RPB on failure. For fail-stop errors, (*e.g.*, crashes, assertion failures), Aviso preserves the RPB before the program terminates. Non-fail-stop errors require Aviso to monitor for failure conditions, and save the RPB when they occur. The way to detect non-fail-stop errors depends on the symptom. Identifying arbitrary failures in general is outside the scope of the work, but simple solutions often work well; *e.g.*, validating output often works – in our tests with the Apache web server Aviso watched for log corruption.

4. Generating SHIELDs

There are two steps to generating SHIELDs: (1) Identifying all interesting event sequences from the final moments of a failing run; and (2) Assembling all possible SHIELDs compatible with those event sequences. After a failure, Aviso enumerates all event sequences in the RPB that could potentially have led to the failure. For every event sequence, Aviso assembles a *candidate SHIELD*. As described in Section 2.3, a SHIELD can be used by Aviso to prohibit the event sequence that it was based on. If a candidate event sequence represents the cause of a failure, the corresponding SHIELD prevents the failure.

4.1 Enumerating Event Sequences

Aviso extracts two types of event sequences from the RPB: *event triples* and *event pairs*. The goal is to represent a failure that has occurred as a triple or pair. Triples represent interleavings of events that can lead to atomicity violations. Pairs represent ordering errors, and also some atomicity errors.

Event Triples Aviso enumerates event triples, $(A1, B, A2)$, such that $A1$ and $A2$ were executed by the same thread, and B by a different thread. We consider triples in which $A1$ preceded B, and B preceded $A2$, but were not necessarily consecutive. The motivation behind considering triples is that they represent situations in which event B interleaves events $A1$ and $A2$. Such an interleaving is an atomicity violation if the programmer expected $A1$ and $A2$ to be atomic. Aviso looks at the RPB from a failing run, so if the atomicity violation occurred, and the program failed when event $A2$ executed, the events making up the atomicity violation will appear as a triple in the RPB. To limit the number of triples, we only include triples for which $A1$ and the $A2$ are within 10 events of one another in the RPB.

Figure 3(a) shows an event triple in the RPB that represents an atomicity violation bug. In the example, `<E 1>` and `<E 3>` should be atomic. However, the sequence shown in the RPB shows that `<E 2>` interleaved them. In the ($A1$, B, $A2$) triple that is produced is, $A1$ is `<E 1>`, $A2$ is `<E 3>`, and B is `<E 2>`.

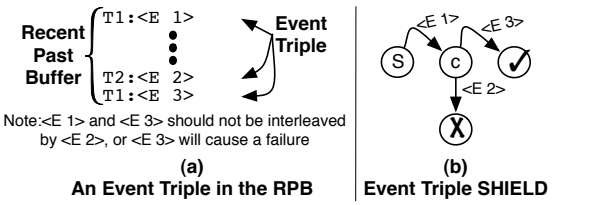

Figure 3. Mapping event triples in the RPB to a SHIELD. (a) shows a triple in the RPB and (b) shows the SHIELD it yields.

Event Pairs Aviso considers pairs of events in the RPB, (B, A), that were executed by different threads. Event pairs can represent ordering violations and some atomicity violations. We select pairs of events under the constraint that between B and A no event was executed by the thread that executed B. Note that between events in a pair, other uninvolved threads may execute other unrelated events. Also, we only consider event pairs separated by fewer than 10 events in the RPB.

Ordering Violations as Event Pairs An ordering violation bug involving two events, A and B, that should execute in A-B order causes a failure when B precedes A. Recognizing the failure with an event pair is a challenge because the execution may fail at B, and event A may never execute. To deal with such situations, Aviso relies on the presence of a third event, C, executed by the same thread as A. The key is that C executes just before A would, and is added to the RPB. When B executes and the failure occurs, C is in the RPB, followed by B. If A had executed, it would have immediately followed C. Hence, when C is followed by B in a failing run, it is an indication that the incorrect ordering of B and A is likely to have occurred.

Figure 4(a) illustrates how an event pair represents an ordering violation. The figure shows a snippet of the RPB from a failing execution. `<E 3>` should precede `<E 2>`, but did not, causing a failure when `<E 2>` executes. `<E 1>` is an event executed by the thread that would have executed `<E 3>`. Aviso identifies (`<E 1>`, `<E 2>`) as an event pair when it analyzes the RPB.

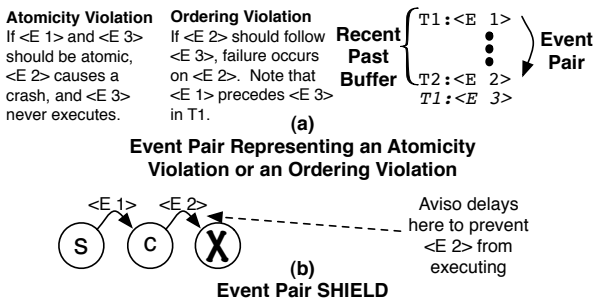

Figure 4. A SHIELD built from an event pair. (a) shows a snippet of the RPB after a failure, and describes how an event pair maps to different types of errors. Note that is not in the RPB, because the failure occurred at `<E 2>`. (b) shows the SHIELD based on the event pair indicated.

Atomicity Violations as Event Pairs Event pairs can also represent atomicity violations. Consider a region of code that the programmer assumed would be atomic that begins with event B, and ends with some other event, C. Recall that if A violates the atomicity of B and C, and the program fails at C, Aviso will capture the bug as a triple, (B, A, C). However, if A violates the atomicity of B and

C, and the program instead fails at A, C never executes. If C never executes, it will not appear in the RPB, making it impossible for Aviso to represent the failure as a triple. However, note that the failure is likely to occur whenever B is followed closely by A. Aviso represents the execution of B and A in succession as a pair, (B,A), and uses that pair to represent the atomicity violation.

Figure 4(a) illustrates how Aviso represents atomicity violations as event pairs. A snippet of the RPB is shown that contains <E 1> and <E 2>. Beneath the RPB <E 3> is shown. The execution failed when it executed <E 2>, preventing <E 3> from being added to the RPB. <E 1> and <E 3> should have executed atomically, but <E 2> interleaved between them. The indicated event pair represents the case where <E 2> follows immediately after <E 1> and before <E 3>, manifesting the atomicity violation.

4.2 Assembling SHIELDs

Aviso builds SHIELDs from every event pair and triple it observes in the RPB after a failure. The process for constructing SHIELDs involves mapping events from the pairs and triples to transitions in a SHIELD. We developed a formal description of SHIELDs, but omit them in this paper due to space constraints.

Event Triple SHIELDs The mapping from a triple (<E 1>, <E 2> <E 3>) to a SHIELD is illustrated in Figure 3(b). The first step is to create a start state, marked S. A transition triggered by <E 1> connects S to c, the "center node". c has two outgoing transitions. One of these transitions is triggered by <E 2> and leads to a failure state, marked $\times$. The other is triggered by <E 3> and leads to a non-failing accept state marked $\checkmark$. If a triple represents the cause of the failure, and during the execution its SHIELD is in the c state, attempts to execute the event leading to the $\times$ state will be delayed, to prevent the execution from failing.

Event Pair SHIELDs Figure 4(b) shows the mapping from the pair (<E 1>, <E 2>) to a SHIELD. Recall from Figure 4(a) that such a pair can correspond to an atomicity violation or an ordering violation involving a third event <E 3>. In an atomicity violation, <E 2> should be prevented from following immediately after <E 1>, preventing an interleaving of <E 1> and <E 3>. In the ordering violation, <E 2> should follow <E 3> and <E 1> precedes <E 3> and is executed by the same thread. Preventing <E 2> from executing immediately after <E 1> gives <E 3> a chance to execute before <E 2>, preventing the error. For both ordering and atomicity bugs, the key is that <E 2> must be delayed after <E 1> executes, giving <E 3> the opportunity to execute.

The start state is marked S. There is a transition triggered by <E 1> from S to the center state, c. c has a transition to the failure state $\times$, triggered by <E 2>. After <E 1> executes, executions of <E 2> are delayed to avoid the $\times$ failure state. The delay lets <E 3> execute, avoiding the failure. Delaying <E 1> indefinitely may pose forward progress issues, so Aviso allows <E 1> to proceed after a fixed interval elapses. After the interval elapses, the SHIELD is terminated as though it has reached an accept state.

4.3 SHIELD Generation Example: Transmission

Figure 5 shows a bug in Transmission-1.42. Figure 5(a) shows a failing execution. Thread 2's `assert(h->bandwidth)` fails if it executes before Thread 1 assigns `h->bandwidth`.

Figure 5(b) shows part of the RPB at the end of the execution. Arcs indicate event pairs. The dashed arc is a pair that, when it occurs, leads to failure. Events' call stacks have been omitted from the RPB for illustrative purposes.

Figure 5(c) shows the SHIELD constructed from the dashed arc pair in Figure 5(b). In the example, `session.cpp:278` is <E 1> and `platform.c:222` is <E 2>. `session.cpp:282` should precede `bandwidth:251` and `platform.c:222`, so `session.cpp:282` is <E 3>. Aviso prevents `platform.c:222` from preceding `session.cpp:282`, preventing the failure.

5. Vetting Candidate SHIELDs

After a failure Aviso generates a set of candidate SHIELDs in order to *vet* candidate SHIELDs and isolate ones that prevent failures. Aviso treats each one as a possible fix and *empirically* determines which are most effective by running them and tracking the failure rate. Aviso monitors executions with different SHIELDs, and discards those that do not significantly decrease the failure rate from the program's baseline rate. Over time, the set of SHIELDs converges to those that prevent failure. From this set, Aviso selects the one that most decreases the failure rate. Before vetting, Aviso reduces the number of SHIELDs by pruning SHIELDs that excessively degrade performance. Aviso runs each SHIELD on a test input, and eliminates any with greater than 200% overhead.

Correct Run SHIELD Pruning We developed a technique reduce SHIELD vetting time. Aviso can use non-failing runs to eliminate SHIELDs that are unlikely to be useful. If an event sequence occurs in a correct run, it is less likely to be the cause of the failure. After a failure and before vetting, Aviso collects snapshots of non-failing executions' RPBs at the point where the failure occurred in the failing run. Aviso generates SHIELDs from the failing and non-failing runs' RPBs. Aviso discards SHIELDs from the failing run that were also produced from a non-failing run. We used a set of 100 non-failing executions, as it was adequate to prune most candidate SHIELDs. Snapshots are easy to collect, and so the number of snapshots used can be as large as is necessary for pruning.

Deployment Vetting Vetting is embarrassingly parallel, and can be expedited by leveraging the scale of deployed software. We envision two deployment vetting scenarios. The first scenario is a datacenter, where many software instances run simultaneously. Different instances vet different SHIELDs in parallel. The second scenario is widely deployed user or system software, in which many different users run instances of software simultaneously (*i.e.*, hundreds of thousands of times per day [2]). Different users can vet different SHIELDs in parallel. In this scenario, the framework can be provided by the software vendor, like a crash reporting service.

Shared Failure Avoidance After vetting, the Aviso framework can *share* effective SHIELDs with all instances of a program. Aviso is able to share SHIELDs because instrumentation is done at the source level, so events are the same across program instances.

6. Evaluation

We evaluate Aviso on several dimensions. First, we show Aviso's failure avoidance efficacy. Second, we show that Aviso's overheads are reasonably low. Third, we characterize Aviso's dynamic behavior. Finally, we characterize Aviso's SHIELD vetting process.

6.1 Experimental Setup

System Implementation We built a full implementation of Aviso, including the profiler, instrumenting compiler, runtime system, and the vetting and avoidance-sharing framework. The profiler was built using Pin [10] and the compiler pass was built using LLVM [6]. The rest of the system was implemented from scratch as a shared library.

Benchmarks We evaluated Aviso using several buggy programs. **Transmission-1.42** (Figure 5) is a bittorrent client with a use-before-initialize bug. We used a setup similar to prior work [17], including their patch to make the bug manifest more frequently. **PBZip2-0.9.1** is a compression tool with a use-after-free bug. We tested PBZip2 by compressing a 250MB file. Like Transmission, we patched PBZip2 to increase its baseline failure rate. **AGet-0.4** (Figure 1) is a download tool with an atomicity violation that leads to output corruption. To test AGet we downloaded a 50MB file from the local network, and interrupted the download with a signal to trigger the error. Aviso diagnosed failures by checking for output corruption. **Apache-2.0.48** is a web server with atomicity violations that cause log corruption. To test

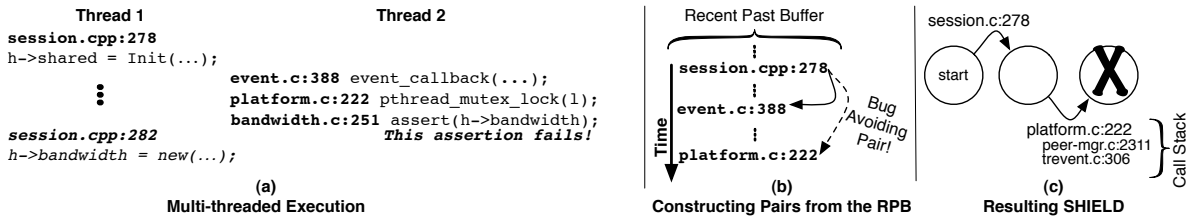

Figure 5. A use-before-initialization bug in Transmission. (a) is a buggy execution. (b) is the RPB just after the failure. Arcs indicate event pairs and the dashed arc is the pair that avoids the failure. (c) shows the failure-avoiding SHIELD Aviso derived from the dashed-arc pair in (b).

	Failure Rate Reduction	Performance Overhead	
		Coll. Only	Coll. & Avoid
Transmission	>1000x	0%	0%
AGet	11.4x	0%	0%
PBZip2	36.0x	0.8%	4.2%
Apache_1	3.2x	3.8%	21.3%
Apache_2	10.5x	3.8%	39.5%

Table 1. A table summarizing Aviso's failure avoidance capability and performance overheads both with and without active avoidance state machines, relative to baseline execution.

Apache, we used `ApacheBench` to issue bursts of 10000 requests from 8 threads for a local `html` page. Aviso diagnosed failures by watching for log corruption.

6.2 Bug Avoidance Efficacy

The main result of our evaluation is that Aviso significantly improved reliability of the applications we experimented with. Column 2 of Table 1 shows the decrease in the failure rate from the baseline for each test program. Notably, the failure in Transmission, which occurs during every baseline execution, never occurred in thousands of executions with Aviso. PBZip2 also improved, failing *36 times* less frequently with Aviso than without.

SHIELD Composability Table 1 also shows that SHIELDs are *composable*. We performed two experiments with Apache. In the first experiment, Apache_1, we ran Apache with no SHIELDs. On seeing a failure, Aviso isolated a SHIELD that reduced the manifestation rate for that failure by 3.2X. In the Apache_2 experiment, we allowed Apache to run under Aviso with the Apache_1 SHIELD. When a second failure occurred, Aviso isolated another SHIELD. Together, the SHIELDs reduced the failure rate by more than 10x.

Remaining Failures The data show that Aviso reduces failure rates, eliminating the effects of the bug in Transmission. However, in other cases failures still occur. There are several reasons failures persist. First, there are often many ways a program can manifest a failure – programs often have more than one bug. Second, Aviso prunes some events online. Under some schedules, important events may be discarded. Third, Aviso is a best-effort approach: Aviso allows delayed events to proceed eventually to preserve progress and performance.

6.3 Performance

Table 1 shows that Aviso's runtime overhead is low. Column 3 is the overhead of event monitoring only. The overhead ranges from negligible in AGet and Transmission to 3.8% for Apache. These results show that non-failing programs will incur little, if any, performance overhead. Column 4 shows the overhead of monitoring and avoidance.

When using SHIELDs to avoid failures, Aviso imposes no measurable overhead on AGet or Transmission because they are network bound. In addition, Transmission's overheads are low because SHIELD instantiation checks occur rarely – about once every 7000 events – because the failure-avoiding SHIELD involves only rare events. PBZip2's overhead was 4.2% because its failure-avoiding SHIELD starts with an event that only executes during shutdown. Hence, SHIELDs do not need to be activated, and impose little overhead. For Apache_1, the overhead was around 21%. Apache_2, with its pair of failure-avoiding SHIELDs, saw overhead around 40%. These overheads are higher than for the other applications, but still low enough for deployment. Furthermore, failures are 10 times less frequent with Aviso, so the overheads may be amortized by a decrease in downtime and time spent patching software.

6.4 Characterization

We instrumented Aviso to characterize its dynamic behavior. Due to space constraints, we omit the full details of our characterization study, and provide highlights here.

Event Behavior First, we characterized Aviso's dynamic event behavior. Our data show that events occur frequently. On average 64% of events occur within 10μs of one another and are discarded. In Apache and Transmission, more than 80% were discarded. On average, 19% are truncated to a single return address. However, as Table 1 indicates, Aviso reduces failure rates for these programs, so the reduced set of events being collected is adequate.

We also characterized the proportion of synchronization and sharing events, which varied across applications. For AGet, there are about two synchronization events for each sharing event, corresponding to the program's inner loop, which has a lock acquire and release, and an access to shared data. Apache has thousands of sharing events per synchronization event, corresponding to many (often incorrect!) unsynchronized acceseses to shared data. PBZip2 has far fewer overall events than Apache or AGet (10s vs. 100,000s), because it rarely shares due to the structure of its parallelism.

SHIELD Behavior SHIELD activation checks occur whenever an event involved in any SHIELD executes. In general SHIELD instantiation checks and activations are infrequent. AGet had about 8,600 SHIELD instantiation checks, and about half led to activations. In Apache, a small fraction of events led to instantiation checks – around 10 per 10,000 non-discarded events. Of these, about 2 per 10,000 led SHIELD activations. For Transmission, 0.17% of non-discarded events led to a SHIELD activation check, and only 1 resulted in an SHIELD activation. The maximum number of simultaneously active SHIELDs for each application is also small. Apache saw the most, with around 28 – this corresponds to thread count (25 worker threads, plus several listener and server threads), suggesting about one active SHIELD per thread. Other applications saw similar numbers.

6.5 Vetting

Table 2 characterizes vetting. Columns 2 and 3 show the number of triples and pairs in the RPB for each application. Column 4 shows the number of SHIELDs not pruned due to excessive overhead. Column 5 shows the CPU time to vet SHIELDs.

The number of candidate SHIELDs produced after a failure is application-dependent. The more interleaving in the RPB, the more pairs and triples will be produced. Apache's RPB has many interleaved events from many threads, resulting in many candidate SHIELDs. PBZip2 fails on shutdown, when most events are executed by the main thread, leading to little interleaving and few candidate SHIELDs. Few SHIELDs were pruned due to low performance – only a handful of Apache's thousands of SHIELDs. AGet saw the

App	Trip.	Pair	NoHang	Time
Transmission	60	110	170	28m
AGet	73	22	38	2h54m
PBZip2	12	160	172	10h53m
Apache_1	2811	4849	7557	70h27m
Apache_2	1250	2449	3690	76h22m

Table 2. A table characterizing SHIELD vetting behavior.

	Failure Rate	Vetting			
App	Reduction	Trip.	Pair	NotPruned	Time
Transmission	>1000x	98	60	6	5m
AGet	20x	177	49	57	2h7m
PBZip2	5.3x	234	222	82	3h49m
Apache	5.0x	3232	4231	62	6h48m

Table 3. A table characterizing the use of correct runs in vetting.

most pruned, with 57 SHIELDs being eliminated. The main reason for the decrease is that all threads run the same loop. Some SHIELDs lead to situations where threads' iterations of the loop are serialized, degrading performance. The time to vet varied from minutes (Transmission) to days (Apache). Vetting is parallelizable, so with many instances (*i.e.,* hundreds), vetting will be fast. The cost of vetting is likely less than the cost to manually fix a program [12], which may include the cost of rushed, incorrect "fixes" [7]. Running with a SHIELD gives developers time to carefully write and test a fix.

6.6 Using Correct Run Information

We performed a complete second set of experiments to evaluate the correct run pruning optimization described in Section 5 to show that it reduces vetting time and does not decrease avoidance. For each application, we collected RPBs from 100 correct runs. We removed all SHIELDs from the set produced from the failing run's RPB that were produced from any of the correct runs' RPBs. We vetted the remaining SHIELDs to isolate the most effective. Table 3 summarizes our findings. Column 2 shows the reduction in failure rate for the most effective SHIELD. Columns 3 and 4 show the initial numbers of pair and triple SHIELDs. Column 5 shows the number of SHIELDs remaining after correct run pruning. Column 6 shows the vetting time for the remaining SHIELDs.

The data show that correct run pruning eliminates a majority of SHIELDs to be vetted – *e.g.*, 99.2% for Apache. The data also show that with this optimization, effective SHIELDs in the set to vet are preserved, and lead to reduced failure rates. For Apache and AGet, the most effective SHIELDs reduced failure rates by more than in our initial experiments. Interestingly, for both applications the failure rate reductions came from SHIELDs that did not show up during our initial experiments. Transmission saw the same failure rate as in the initial tests, and the most effective SHIELD was very similar to the one in our initial experiment. PBZip saw a 5.3x decrease in failure rate in this experiment – less than the first experiment, but still a considerable improvement over the baseline. The data show using correct runs cuts vetting time without compromising avoidance.

7. Related Work

There has been a great deal of recent work on techniques dealing with software failures. Due to space constraints, we focus here on work that deals with concurrency-related failures.

Most related is Loom [16], a system for patching concurrency bugs in running programs. Loom is like Aviso in that it aims to prevent failures between when a bug's symptom appears, and when a patch is released. Aviso differs from Loom in an important and fundamental way: Loom requires the user to understand the *cause* of a failure well-enough to write a work-around. Aviso is automated, requiring nothing from the programmer in most cases to produce a work-around SHIELD. For some non-fail-stop errors, the user must recognize a bug's *symptom*, which is easier than understanding the cause.

There has been a lot of work on atomicity bugs [3, 8, 9, 14, 15]. Isolator [14] and ToleRace [15] prevent single-variable atomicity violations, but do not handle the broader class of failures addressed by Aviso. AFix [3] produces bytecode patches that fix atomicity bugs. AFix is like Aviso in that it eliminates the need to think about a failure's cause. Unlike Aviso it is limited to atomicity errors. AtomAid [9] and ColorSafe [8] address single- and multi-variable atomicity bugs. These systems are unlike Aviso in that they only handle atomicity bugs, and need hardware. Other systems have proposed using hardware support to force executions to adhere to tested schedules [17, 18]. These systems are similar to Aviso in that they aim to prevent concurrency-related failures. They differ in that they use hardware, and can ensure reliability only in tested situations. Aviso relies on information collected from a single failing run, and can apply it in any situation to avoid failures. Dimmunix [5] and Communix [4] provide automated deadlock immunity for Java programs. These systems are similar to Aviso in that they identify and avoid failures mostly automatically, and Communix systems share failure avoidance capability. These systems are limited in that they only address deadlocks.

Determinism [1, 11, 13] enforces one event interleaving in every execution. Aviso also enforces event orderings, but unlike determinism, only over events involved in SHIELDs, not entire executions. Aviso differs in that it does not restrict the interleaving of the entire program. Instead, all possible executions are permitted, except where restricted by SHIELDs.

8. Conclusions

In this work we presented Aviso, a system for automatic avoiding concurrency-related failures. Aviso selectively monitors program events, and when a failure occurs, Aviso finds event sequences that may have been responsible. Aviso builds and vets candidate SHIELDs during subsequent executions. SHIELDs that avoid failures are distributed to other software instances to share failure avoidance. We empirically showed that Aviso effectively avoids failures in several real-world programs. We also showed that Aviso operates with high-performance, imposing overheads low enough for deployed software.

Acknowledgments

We appreciate the useful feedback from our anonymous reviewers. We also thank Dan Grossman, Laura Effinger-Dean, Tom Bergan, Pete Hornyack, and Todd Schiller for their wonderful suggestions on improving the paper. We thank Anthony Fader and Adrian Sampson for their superb early contributions to this work.

References

[1] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[2] T. M. Chilimbi, A. V. Nori, B. Liblit, K. Vaswani, and K. Mehra. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[3] G. Jin, L. Song, W. Zhang, S. Lu, , and B. Liblit. Automatic atomicity-violation fixing. In *PLDI*, 2011.

[4] H. Jula, P. Tozun, and G. Candea. Communix: A collaborative deadlock immunity framework. In *DSN*, 2011.

[5] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, pages 295–308, 2008.

[6] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[7] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, 2008.

[8] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-Variable Atomicity Violations. In *ISCA*, 2010.

[9] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.

[10] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[11] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.

[12] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. `http://www.nist.gov/director/prog-ofc/report02-3.pdf`, 2002.

[13] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[14] S. Rajamani, G. Ramalingam, V. Ranganath, and K. Vaswani. Isolator: Dynamically ensuring isolation in concurrent programs. In *MICRO*, 2009.

[15] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *IEEE Transactions on Computers*, 2011.

[16] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[17] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA*, 2009.

[18] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, 2010.