Cooperative Empirical Failure Avoidance for Multithreaded Programs

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

Abstract

Concurrency errors in multithreaded programs are difficult to find and fix. We propose Aviso, a system for avoiding schedule-dependent failures. Aviso monitors events during a program's execution and, when a failure occurs, records a history of events from the failing execution. It uses this history to generate schedule constraints that perturb the order of events in the execution and thereby avoids schedules that lead to failures in future program executions. Aviso leverages scenarios where many instances of the same software run, using a statistical model of program behavior and experimentation to determine which constraints most effectively avoid failures. After implementing Aviso, we showed that it decreased failure rates for a variety of important desktop, server, and cloud applications by orders of magnitude, with an average overhead of less than 20% and, in some cases, as low as 5%.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.2.5 [*Testing and Debugging*]: Error handling and recovery

General Terms Algorithms, Reliability

Keywords concurrency, cooperative failure avoidance

1. Introduction

Concurrency errors in multithreaded programs can lead to non-deterministic *schedule-dependent failures*, which are failures that arise from interactions between threads that are unforeseen by programmers during development. Such interactions can be characterized by sequences of synchronization operations and accesses to data that the threads share. Atomicity violations, a common type of schedule-dependent failure, occur when one thread's access to shared state is incorrectly permitted to interleave between a pair of accesses in another thread. Ordering violations, another type of schedule-dependent failure, take place when operations in different threads occur in an order that leads to a failure.

Most prior work dealing with concurrency errors has focused on their *detection*, which typically involves identifying patterns of memory accesses specific to certain bug types. Unfortunately, even with state-of-the-art tools, concurrency bugs remain in deployed code and cause costly, schedule-dependent failures.

Recent research is exploring the *avoidance* of schedule-dependent failures. Avoidance work seeks to determine the sequences of operations most likely to lead to buggy behavior and to prevent those sequences from executing. This general idea has had several incarnations. For example, prior work proposed the use of hardware to prevent atomicity violations [12, 13] and to bias execution towards tested behavior [27, 28]. Other avoidance work used memory-protection-based techniques [21, 22] and relied on programmer annotations to modify strategic points in a program [26].

These recent efforts have made progress toward more reliable concurrent systems, but they fall short in several ways. Some require significant programmer effort, which may be better spent patching the faulty program. Many cater only to certain types of bugs, and several require complex, expensive changes to hardware.

Failure avoidance systems face many complex challenges. Systems must monitor program execution to collect the information needed to identify likely failures. This monitoring must be efficient because failure avoidance is most useful in deployed systems, where performance is paramount. Systems also require precise techniques to identify failures using the collected data and mechanisms to influence the program's execution to avoid failures; as with monitoring, such mechanisms must be efficient. Furthermore, systems should avoid failures without requiring modifications to hardware or changes to the way programmers write their programs.

We overcome these challenges in this work. Our main contribution is a *novel, automated technique for avoiding schedule-dependent failures*. We develop an efficient system for collecting relevant program events at run-time in deployed software. When a program instance fails, we use the information collected by our system to generate hypotheses about what caused the failure. Then, leveraging the fact that we often have a number of deployed instances of the same software, we develop a predictive statistical model and an empirical framework to identify which hypothesis is most likely to be correct. Based on that hypothesis, we influence future program executions by perturbing the thread schedule to avoid subsequent failures.

We implement these ideas in Aviso, a distributed framework that coordinates many deployed instances of a program to cooperatively learn how to prevent failures. Our system works on commodity hardware with minimal impact on software and the development process. Our evaluation on several real-world desktop, server, and cloud applications shows that Aviso effectively and efficiently improves software reliability. In one test, Aviso reduced the failure rate of a buggy version of Memcached, a popular key-value store, by two orders of magnitude. Fixing the bug took developers about a year and their fix imposed about 7% performance overhead [1]. Aviso worked without developer intervention and in minutes found an effective constraint that imposed only a marginally higher overhead of about 15%.

The remainder of this paper describes Aviso in greater detail. Section 2 provides background that explains how Aviso avoids schedule-dependent failures. Section 3 reviews Aviso's design

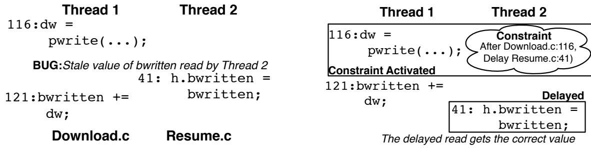
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$10.00

goals and architecture, while Section 4 relates how Aviso decides which events to monitor during an execution and how events and failures are to be monitored. We describe how Aviso generates candidate constraints after a failure in Section 5 and how it determines which candidate constraint effectively prevents failures as well as inter-system constraint sharing in Section 6. Section 7 provides implementation details. Sections 8–10 evaluate and characterize Aviso, contrast it with prior work, and summarize our findings.

2. Schedule-Dependent Failures

A program's execution experiences a schedule-dependent failure when threads execute a particular sequence of events that leads to a crash, contract violation (*e.g.*, assertion), or state corruption. Schedule-dependent failures are the result of programming errors, such as absent or incorrectly used synchronization.

Figure 1(a) illustrates how a programming error leads to a failure using an example from AGet-0.4, a multithreaded download accelerator. Figure 1(a) shows a snippet of a failing multithreaded execution. The failure is an atomicity violation: Thread 1 writes bytes to a file (`pwrite(...)` on line 116) and then adds the number of bytes written to the file to a shared counter (incrementing `bwritten` on line 121). The failure occurs when Thread 2 asynchronously reads the value of `bwritten` on line 41, between the call to `pwrite()` and the increment that follows. The programmer has omitted synchronization operations in Thread 2, permitting that thread to read an intermediate value. Note that if Thread 2's read was delayed, and Thread 1's update was allowed to proceed, the failure could have been avoided.



(a) **A failing execution.** The failure occurs when Thread 2 reads a stale value of `bwritten`.

(b) **Avoiding the failure.** The constraint delays Thread 2, reordering operations to avoid the failure.

Figure 1. A schedule-dependent failure in AGet-0.4.

2.1 Bug Depth

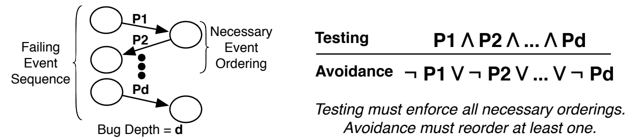
Recent concurrency testing work [3] formally characterized concurrency errors using the notion of *bug depth*. A bug's depth is the minimum number of pairs of program events that must occur in a particular order for that bug to manifest a failure [3]. Note that a single *bug* can lead to different *failures* under different execution schedules. A different set of event pair orderings is necessary to manifest each different failure and a failure occurs only if all its necessary orderings are satisfied. Typically, most schedules do not satisfy all orderings required to cause any particular failure, so executions usually do not fail.

This fact poses a key challenge to multithreaded testing. To expose a bug during testing by causing a failure, systems must enforce a number of orderings on the execution schedule that is greater than or equal to the bug's depth. The larger the bug's depth, the more work is needed to ensure that a failure occurs [3]. The bug in Figure 1(a) has depth 2 because two pairs of events must execute in a particular order for the failure to manifest: Thread 2's read must follow Thread 1's `pwrite` call and must precede Thread 1's `bwritten` update. If the first ordering were not upheld, Thread 2 would correctly see the value of `bwritten` before any of Thread 1's operations. If the second ordering were not upheld, Thread 2 would correctly see the result of Thread 1's operations.

2.2 The Avoidance-Testing Duality

In this work, we make the observation that a failure caused by a bug of *any* depth can be avoided by perturbing just *one* pair of events in the sequence that leads to the failure. Given a chain of pairs of events that must be ordered to manifest a failure in testing, perturbing the schedule to reorder any of the pairs "breaks the chain", preventing that failing schedule from occurring. This observation suggests a duality between testing for schedule-dependent failures and avoiding them: testing requires enforcing *all* of a conjunction of pair orderings to exercise a failing schedule; avoidance requires reordering events in *any* of those pair orderings to avoid a failing schedule. Avoiding a particular failing schedule leads the execution to a new schedule. Avoidance is successful if the new schedule does not lead to a failure, which is likely the case since there are typically significantly more failure-free schedules than schedules with failures.

Figure 2(a) illustrates bug depth. The circles are program events and the arrows represent pair orderings necessary to lead to a failure. There are d orderings, so assuming the figure shows the fewest possible orderings for the bug to cause a failure, the bug has depth d. Figure 2(b) contrasts testing and failure avoidance. Exposing a bug during testing requires satisfying a conjunction of d pair orderings, whereas the failure is avoided if a single pair of events is reordered.



(a) **Pair orderings necessary for the failure to occur.** There are d orderings, so the bug responsible for the failure has depth d (assuming the figure shows the fewest orderings for the bug to cause a failure).

(b) **Contrasting testing with failure avoidance.** Testing aims to expose bugs, so *all* orderings must be satisfied. Avoidance aims to prevent failures by reorderings the events in *any* ordering.

Figure 2. The Avoidance-Testing Duality.

3. System Overview

This section provides an overview of Aviso's system architecture and design constraints. We then walk through Aviso's failure avoidance mechanism with an example, and explain how Aviso facilitates *cooperative*, *empirical* failure avoidance in a community of software instances.

3.1 System Architecture

Figure 3 presents Aviso's four components: (1) the profiler, (2) the compiler, (3) the runtime system, and (4) the framework.

Profiler and Compiler. After development, the programmer runs Aviso's profiler, which determines what program operations to monitor. The profiler sends an event profile to Aviso's compiler. The compiler uses the event profile to add event-handling runtime calls to the binary and links the binary to the Aviso runtime to produce and Aviso-enabled executable.

Runtime. The Aviso runtime system monitors and keeps a short history of events during program execution. The runtime also watches for failures and alerts the framework when they occur. Periodically during execution and when execution fails, the runtime sends its event history to the framework. The runtime can also perturb the execution schedule using *schedule constraints* (see below) that it receives from the framework.

Framework. Aviso-enabled executables run in the Aviso framework and the two communicate via a simple messaging API. The framework collects event histories and failure information sent by the runtime. It generates schedule constraints from this information and sends them to the runtime when Aviso-enabled executables start running. The framework selects constraints to send using a statistical model that predicts which constraint is most likely to avoid failures; it builds the model using aggregated history and failure information. By aggregating information from and sending constraints to many program instances, the framework enables program instances to cooperatively avoid failures.

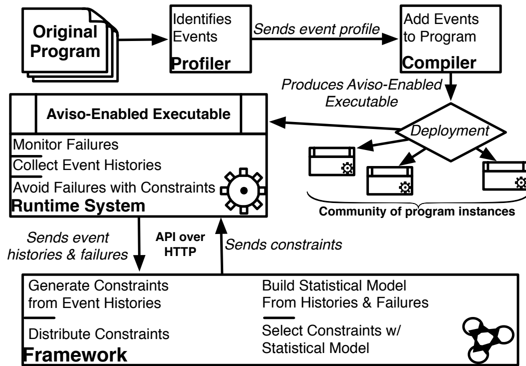


Figure 3. Aviso's components. The compiler and profiler find and instrument events. The runtime system monitors events and failures and avoids events. The framework generates constraints, selects likely effective constraints using a statistical model and shares effective constraints in a community of software instances.

3.1.1 Design Requirements

Our goal is to build a system that is general enough to avoid failures due to a broad class of concurrency errors in a deployment setting. This goal presents us with conflicting design constraints: For the sake of generality, the runtime should monitor as many program operations as possible, to capture a large variety of failure behaviors. However, monitoring imposes a time and space overhead, and building a system intended for use in deployment necessitates high performance. Aviso should find effective constraints quickly and permit as few failures as possible. To do so Aviso must leverage *prediction* to identify effective constraints without having to directly observe their impact on the program's behavior. Aviso is additionally constrained because programmer time is valuable and understanding and fixing concurrency errors is difficult, error-prone, and time consuming. Our system should require as little as possible from the programmer.

3.2 Avoiding Failures with Schedule Constraints

Aviso leverages the avoidance-testing duality introduced in Section 2.2 to avoid failures. Aviso uses *schedule constraints* to perturb an execution's thread schedule with the goal of reordering events in the execution whose original order leads to a failure.

A constraint is based on a pair of events observed by Aviso in a failing program execution. It is "activated" when the first event in its pair executes. When a constraint is active and a thread tries to execute the second event in the constraint's pair, Aviso delays the thread, reordering some events in the execution. If the original order of the reordered events was necessary for a failure to occur, reordering the events will avoid the failure. Such reorderings are the key to Aviso's failure avoidance mechanism. We provide more details on avoidance in Section 5.2.

Figure 1(b) shows how Aviso uses a constraint to avoid a failure. The constraint is made from a pair of events: the first event is Thread 1's `pwrite` call at Download.c, line 116, and the second event is Thread 2's read of `bwritten` at Resume.c, line 41. When the first event is executed, the constraint is activated. While the constraint is active, Thread 2 attempts to execute the second event, and it is delayed. During this delay, Thread 1's `bwritten` update executes atomically with its `pwrite` call, preventing the failure. Later, when Thread 2 resumes execution, it reads the correct value of `bwritten`.

3.3 Cooperative Empirical Failure Avoidance

Aviso is an *empirical* failure avoidance system. When a failure occurs, it generates a set of constraints from its event history (see Section 5). Each constraint is a *hypothesis* about how to prevent the failure. Aviso decides which constraints are most likely to avoid failures and instructs program instances to use those constraints. It is an *empirical system* because it uses a combination of predictive statistical modeling and experimentation to select effective constraints. As Aviso observes more execution behavior, it refines its model, improving its selections. The details of constraint selection are described in Section 6.

Aviso is also a *cooperative* failure avoidance system. It leverages communities of computers running the same program to select effective constraints in two ways. First, Aviso's statistical model is built using information drawn from a community of program instances. Second, Aviso distributes constraints to all members of a community. A constraint that consistently avoids failures for some members can be distributed to other members, sharing its benefit.

4. Monitoring Events and Failures

Aviso's profiler identifies program operations relevant to concurrent program behavior. The compiler inserts event-handling runtime calls into Aviso-enabled executables before each operation. Aviso works on deployed programs, so determining which operations should be treated as events determines the overhead of event handling. Aviso uses static and dynamic analyses to prune the detected set of events. Aviso's runtime traces events during execution and monitors for failures.

4.1 Identifying Relevant Program Events

Aviso focuses on concurrency errors, so we restrict our attention to events related to concurrency. There are three types of events that Aviso monitors: (1) synchronization events, (2) signal events, and (3) sharing events. *Synchronization events* are lock and unlock operations, thread spawn, and join operations that can be identified by matching synchronization library (*e.g.*, `pthread`) calls. Aviso can handle other types of synchronization, as well (*e.g.*, CAS) if the programmer identifies them as sychronization.

Signal events are functions that handle signals. They are of interest because signals may be delivered and handled asynchronously. Signal events are 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 they cannot be identified by looking only at syntactic properties of a program. Instead, Aviso identifies sharing events using a *sharing profiler* before application deployment, *i.e.*, during testing. The sharing profiler monitors threads' accesses to shared data. When a thread accesses data that has been accessed by another thread during the execution, the operation the thread is executing is considered to be a sharing event. Sharing events are reported by the profiler

and inserted into the deployment binary by Aviso's instrumenting compiler. Aviso identifies events by their instruction address and the call stack when the event occurs.

4.2 Pruning and Instrumenting Events

Handling events too frequently leads to high performance overheads. To mitigate that issue, we use two techniques to reduce the number of handled events. First, our instrumenting compiler uses *dominance pruning* to eliminate instrumentation of some events. Second, our runtime system uses *online pruning* to limit the number of events that are handled.

Dominance Pruning (Static) Analysis. When compiling a function and before optimization, Aviso's compiler computes the set of dominators for each instruction. We use the computed dominance relationships to prune the set of candidate events. Given a pair of events (p, q), if p dominates q, then for every execution of q, there was a prior execution of p. Hence, tracking only p captures nearly as much information as tracking both p and q. In this situation, we remove q from the set of candidate events. If p and q are far apart in the code, dominance pruning might discard useful events. However, our analysis did not pose problems in our experiments for several reasons. First, dominance pruning does not apply to synchronization events, and synchronization events often occur near sharing events. Second, we identify sharing events using profiling, which is approximate; dominance pruning makes the approximation only slightly less precise. Third, dominance pruning operates at function granularity, limiting the distance between p and q to the length of a function at most. Fourth, if events are far apart, the dominance relation still conveys information about the interleaving of events along certain control flow paths. This information is less precise but still useful to prevent failures.

Online Pruning. To further reduce overheads, Aviso uses *online pruning* to adaptively reduce the number of events handled. During execution, Aviso tracks the interval between consecutive events. If two events occur within 1μs of one another, they likely encode redundant information, and Aviso discards the second of the two. Discarding events is, in effect, dynamically coalescing a sequence of events that occur within 1μs of one another into a single event, represented by the first in the sequence.

4.3 Tracing Important Program Events

When generating schedule constraints, Aviso focuses on pairs of events that occurred in a single failing program execution. When an execution fails due to a concurrency bug, the event sequence that caused the failure must have occurred during that execution. Aviso focuses on program events that occurred just before the failure. These events are likely to be related to the failure because *some* code point must have trigger the failure (*e.g.*, caused a crash, emitted buggy output, violated a contract, *etc.*); these events must have occurred occurs shortly before the symptom of the failure was manifested. This observation suggests that a backward scan over a trace of events from the point of failure is likely to encounter the events involved in the failure.

We therefore designed the Aviso runtime to maintain a totally ordered history of events recently executed by any thread, called the *Recent Past Buffer*, or RPB, for the execution. The RPB is a fixed-size queue; the size could vary across implementations, but it should be on the order of hundreds of events. We used an RPB that holds at most 1000 events from each thread. Across most of our experiments, we saw an average event frequency of around 500μs, so each thread's RPB covers about the last 0.5s of its execution. Half a second is likely to be long enough to capture events related to a failure, as prior work suggests that such events often occur over short windows of execution [10, 11, 13]. When an event is executed, 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 and is likely to include the events that led to the failure.

4.4 Monitoring Program Failures

For crashes and assertion failures, the runtime preserves the RPB before the program terminates. For other failures, Aviso monitors for *ad hoc* failure conditions and preserves the contents of the RPB when failure conditions are met.

The best way to detect non-crash failures depends on the failure's symptom. Identifying arbitrary failures automatically is a difficult problem, and doing so comprehensively is outside the scope of this work. However, simple solutions often work well; *e.g.*, validating output is often adequate. In our tests with Memcached, we added an assertion that encodes a simple data structure invariant, preventing the use of deallocated storage. Section 8.1 describes our experience adding failure monitors to programs for the subset of our tests that required it.

In general, given an error report that describes a failure's symptom, an *ad hoc* failure monitor can be added to the Aviso framework to handle any failure diagnosis criteria. Using failure monitors is not always necessary – Aviso deals with fail-stop errors by default. When necessary, adding failure monitors is less risky and onerous than patching code or writing a workaround [26].

5. Generating Constraints and Avoiding Failures

After a failure, the framework examines the RPB and enumerates event pairs that could potentially have led to the failure. For each pair, it generates a *candidate constraint* that perturbs the thread schedule around the events in the pair. A candidate is *effective* if its perturbation avoids a failure. The framework selects candidate constraints to make available to future program executions that can use them to avoid failure. In Section 6 we discuss how Aviso selects effective candidates.

5.1 Generating Candidate Constraints

We take a straightforward approach to selecting event pairs to generate constraints. The framework considers pairs of events in execution order in the RPB, (B, A), that were executed by different threads. It selects these pairs 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.

Figure 4 illustrates the process of enumerating event pairs. Notice that Thread 1's first execution of E is not part of a pair because it is immediately followed by B in the same thread. Also notice that Thread 2's F and Thread 3's X form a pair in spite of their separation by Thread 1's executions of E.

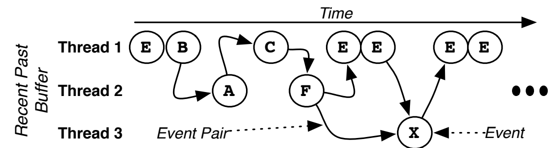


Figure 4. Enumerating pairs from a failing execution's RPB. There are three threads, and time proceeds left to right. Circles are events, and arcs between events are event pairs. Arcs for duplicate pairs are omitted. The figure shows a single 10-event window of events, but selection occurs for all 10-event windows.

To limit the number of constraints generated, we rely on the assumption that events that comprise effective constraints occur

within a short window; we consider only event pairs separated by fewer than 10 events in the RPB. This assumption is reasonable for several reasons. First, prior work on finding and avoiding concurrency bugs [8, 12, 13, 18, 24, 25] suggests that the events involved in schedule-dependent failures often occur within a short window of the program's execution (*i.e.*, hundreds or thousand of instructions). Second, each event in the RPB represents a span of the program's execution, not a single instruction. Due to our online pruning approach (Section 4.2), two consecutive events in the same thread are at least 1μs apart, meaning that each event can represent thousands of instructions. Hence, a 10-event window covers a part of the execution large enough to contain useful event pairs.

5.2 Avoiding Failures

Each event pair corresponds to a schedule constraint. The first event in the pair is the constraint's "activation event", and the second event is the "delay event".

When a program instance starts, the framework makes a set of constraints available to the program instance. Every constraint starts as inactive. Inactive constraints have no effect on the program's execution. When a constraint's activation event is executed, the constraint is *instantiated*, and added to a set of active constraints. The runtime system records the ID of the thread that executed the activation event as the "activator" in the constraint instance. A thread may have at most one instance of a constraint active, but if several different threads execute a constraint's activation event, each thread will instantiate its own instance of the constraint. Figure 5 illustrates the constraint activation process.

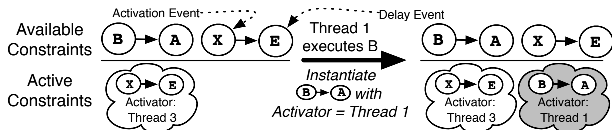


Figure 5. Constraint Activation. *Available constraints* are those that Aviso has made available to the execution. *Active constraints* are constraint instances that have been instantiated and can trigger delays. The large, central arrow signifies Thread 1 executing event B. To the left of the arrow there are no instances of the constraint (B, A); event B is its activation event, so when B is executed an instance of the constraint is added to the Active Constraints set (shaded cloud). Aviso records that Thread 1 is the instance's activator in the instance.

When a thread executes the constraint's delay event, Aviso decides whether to perturb the execution. To do so, it compares the executing thread to the activator of each constraint instance. If the thread executing the delay event is the same as the activator of a constraint instance, Aviso does nothing and execution continues. If it is different, Aviso delays that thread's execution. The delay perturbs the execution schedule by permitting threads other than the delayed one to continue their execution ahead of the delayed event. The reordering of these other threads' events with the delayed event is Aviso's strategy for preventing failures, as we described in Section 3.

5.2.1 Practical Issues with Constraints

There are several practical issues related to pair-based constraints.

Delay Length. Events delayed by constraints cannot be delayed indefinitely without impeding forward progress. Delays must be long enough to reorder events that would lead to failures, but short enough that their impact on performance is tolerable. We empirically determined that $1ms$ achieved this balance well across our benchmarks: any shorter and Aviso was unable to prevent failures in some cases; any longer and performance degraded without improvement in failure avoidance. We show data in Section 8 that further support our choice of delay length.

Composing Constraints. It is important that Aviso not be limited to preventing only one failure due to one bug at a time. Most programs have more than one bug. Each bug may lead to a different failure. To deal with this problem, Aviso can make a collection of constraints available to threads, each with different activation and delay events. When any constraint's activation event is encountered, the executing thread instantiates that constraint. Threads can instantiate multiple different constraints simultaneously to avoid multiple different classes of failures. Section 6 describes how Aviso decides when multiple constraints should be available to be instantiated.

5.2.2 Why Do Event Pairs Make Effective Constraints?

Section 2 discussed the relationship between schedule-dependent failures and bug depth: if we invert one of the d event pair orderings necessary for a bug of depth d to cause a failure, we prevent the failure. In general, Aviso is effective if it generates constraints that reorder such events, like the one in the center part of Figure 6. We now describe how Aviso can use the constraint in Figure 6 to avoid two concrete classes of failures – atomicity violations and ordering violations.

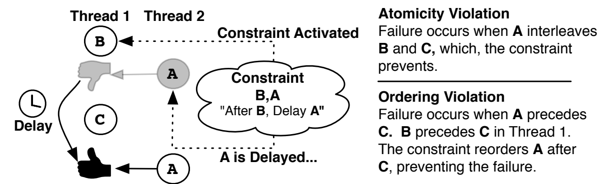
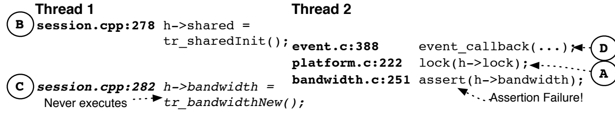


Figure 6. How a constraint avoids a failure. The constraint is shown in the cloud and is made from events B and A; when a thread executes B, the constraint is instantiated. When another thread executes A, it is delayed. The left side shows an execution snippet that can be viewed as both an atomicity violation and an ordering violation.

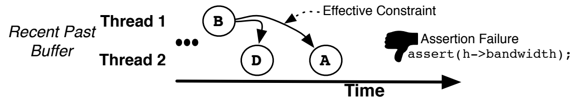
Avoiding Atomicity Violations. Figure 6 shows how constraints avoid atomicity violations. It depicts two threads with Thread 1 executing events B and C, which should not be interleaved by other events. Thread 2 is executing event A. The atomicity violation occurs if A happens between B and C. Note that there are two points in the execution where the failure can occur – just after A executes or just after C executes. In either case, the constraint prevents the failure. When Thread 1 executes B, it instantiates the constraint. When Thread 2 tries to execute A, it is delayed. During the delay, Thread 1 safely executes C. The delay prevents the failure by reordering A after C, rather than between B and C.

Avoiding Ordering Violations. Figure 6 also shows how constraints avoid ordering violations. To view the figure as an ordering violation, assume a failure occurs when Thread 2 executes A before event C.

Avoiding ordering failures is challenging because when the failure manifests, execution may fail just after C or just after A. If the program fails just after A, C will never execute and will therefore not appear in the RPB after the failure, so the Aviso framework will be unable to use it to form a constraint. To handle these failures, Aviso relies on the presence of a third event, B, executed by the same thread as C (Thread 1). In failing runs, B executes just before C would have and is added to the RPB. When A executes and the failure occurs, B is in the RPB followed by C. If



(a) **A failing execution.** Events are identified by the labeled circles.



(b) **The RPB just after the failure.** Arcs indicate the event pairs Aviso enumerates and uses to generate constraints. The dashed arrow indicates that the pair (B, A) corresponds to a constraint that effectively avoids the failure by delaying Thread 2's execution of A until after Thread 1 executes B.

Figure 7. A use-before-initialization failure from Transmission and the constraint that avoids it.

C had executed, it would have immediately followed A. Hence, A following B in the RPB of a failing run indicates that the incorrect ordering of A and C is likely to have occurred.

The constraint in the figure is formed from B and A. When Thread 1 executes B, it activates the constraint. Later, when Thread 2 tries to execute A, it is delayed by the constraint. The delay gives C a chance to execute, preceding A. When the delay expires, A executes after C, avoiding the failure.

5.3 Constraint Generation Example: Transmission

Figure 7 illustrates a failure that can occur when Transmission-1.42 is starting up. Figure 7(a) shows an execution of the events that lead to a failure. The execution fails when Thread 2 reaches `assert(h->bandwidth)` at `bandwidth.c:251` before Thread 1 assigns `h->bandwidth` at `session.cpp:282`. In this situation, `h->bandwidth` is uninitialized, so the assertion fails.

Figure 7(b) shows an RPB snippet immediately after the execution fails. The way the constraint avoids the failure corresponds directly to the ordering violation situation in Figure 6. The constraint delays A (Thread 2's lock acquire at `platform.c:222`), making it execute after C (Thread 1's assignment at `session.cpp:282`). This reordering prevents a failure because `h->bandwidth` is initialized by Thread 1 before the assertion executes.

6. Selecting and Distributing Constraints

After an execution fails, the Aviso framework generates a large set of candidate constraints to assess which can prevent failures. It *selects* the constraints most likely to avoid failures and *distributes* them to new program instances when they start up.

6.1 Selecting Constraints

Aviso selects constraints using a two-part statistical model. The first part of the model is the *event pair model* that represents properties of event pairs, as they occur in correct and failing executions. The second part, the *failure feedback model*, empirically determines which constraints are most effective by tracking each constraint's impact on the program's failure rate. As Aviso progressively observes more failing and non-failing program runs, its models improve, yielding better selections.

6.1.1 Event Pair Model

The event pair model represents each constraint with a vector of features. The value of each of these features is different for each constraint and is derived from execution behavior observed by Aviso in failing and non-failing executions. The magnitude of a constraint's feature vector determines how likely the constraint is to be effective.

There are two main concerns related to the event pair model: (1)Collecting the information that goes into building the model; and (2)Describing and computing the features' values for each pair.

Collecting Model Information The event pair model synthesizes information in RPBs from both non-failing and failing program executions. When Aviso collects an RPB, it updates the event pair model by recomputing each constraint's feature values.

The model uses the information in RPBs from failing program executions. In Section 5 we described how Aviso collects post-failure RPBs to generate constraints. Aviso uses those RPBs to update its event pair model. Aviso also collects RPBs from non-failing program executions. To do so, Aviso samples the state of the RPB very rarely during correct executions. At a uniformly randomized interval between 0.1s and 20s, Aviso interrupts execution, captures the state of the RPB, and uses it to update the event pair model. Note that if Aviso samples an RPB, just before a failure occurs, the RPB may contain events related to the failure. To keep these events out of its set of correct RPBs, Aviso waits a fixed period of 5s before incorporating the correct run RPB into its model. If in the interim the execution fails, the sampled RPB is discarded.

Features Aviso represents each constraint with a vector of three features: *ordering invariance*, *co-occurrence invariance*, and *failure correlation*. Feature values are between 0 and 1. We engineered our feature representation so that larger feature values indicate a constraint is more likely to prevent a failure.

Ordering Invariance (OI) helps identify constraints whose events occur in one order in non-failing runs, but the opposite order in failing runs. Given a constraint built from a pair of events, (B, A), its $OI_{B,A}$ value is:

$$OI_{B,A} = \frac{\sum_{c \in CorrectRuns} f^c_{A,B}}{\sum_{c \in CorrectRuns} f^c_{A,B} + f^c_{B,A}}$$

where $f^c_{x,y}$ represents the number of times the pair x, y appears in the RPB sampled from correct run c. Note that the value of OI is larger if (A, B) occurs much more often in correct runs than (B, A). A large OI value suggests (B, A) is anomalous in correct runs and therefore related to the failure. Hence, perturbing the execution around (B, A) is more likely to avoid the failure.

Co-Occurrence Invariance (CI) identifies constraints whose pairs of events tend not to occur together in non-failing runs, but occur together in failing runs. Given a constraint built from a pair of events, (B, A), its $CI_{B,A}$ value is:

$$CI_{B,A} = 1.0 - \frac{\sum_{c \in CorrectRuns} f^c_{B,A}}{\sum_{c \in CorrectRuns} [\sum_{y \neq A} f^c_{B,y} + \sum_{x \neq B} f^c_{x,A}]}$$

Note that the fraction part of CI is large if B and A occur together frequently in non-failing runs, or if B and A occur with other events infrequently in non-failing runs. Both of these conditions suggest the pair (B, A) is *not* an anomaly in non-failing runs. We invert the sense of the fractional term subtracting it from 1.0. As a result, the value of CI is larger if B and A more often occur in non-failing executions in pairs with different events, rather than with one another.

Failure Correlation (FC) identifies pairs of events that occur frequently in failing executions. Given a constraint built from a pair of events, (B, A), its $FC_{B,A}$ value is:

$$FC_{B,A} = \frac{F_{B,A}}{F}$$

where F is the total number of failing executions and $F_{B,A}$ is the total number of failing executions in which (B, A) occurred at least once. A large FC value suggests that B and A tend to occur consistently in failing executions and are therefore related to the failure; therefore perturbing the execution around (B, A) is likely to avoid the failure.

FC is unlike CI and OI in two ways. First, CI and OI are computed based on RPBs from non-failing executions; FC is computed using data from failing executions. Second, unlike CI and OI, FC is computed using $F_{B,A}$ and F – numbers of executions, rather than numbers of occurrences of pairs (*e.g.*, $f^c_{B,A}$). This is because the frequency of a pair unrelated to a failure in a failing execution may be different because the execution terminated early due to the failure. Such differences act as noise in our model. Instead, our method for computing FC factors out this source of noise.

6.1.2 Failure Feedback Model

The second mechanism Aviso uses to select constraints is the failure feedback model. This model records the failure rate, FR, observed for each constraint. The model also records the failure rate with no constraints available. If the program's failure rate is low (*i.e.*, many non-failing runs, few failing runs) when a particular constraint is active, it is likely that that constraint helps avoid failures more than others.

Every time the program terminates, the failure feedback model is updated. If the program exited normally, Aviso increments the model's record of the number of non-failing runs for all constraints available to the program instance during that execution. If the program fails, Aviso updates the model's record of the number of failing runs.

Dealing with Long-Running Programs To keep the failure feedback model up to date, programs send Aviso a message indicating success or failure when they terminate. However, long-running programs like servers terminate infrequently. If an constraint is effectively preventing failures, the program may run indefinitely. In this case, Aviso might *never* update the failure feedback model to reflect the success of the constraint. To handle long-running programs, we add a facility to Aviso to record "logical time" ticks. To use logical time, we rely on the programmer to add markers to the code that represent progress in the application. Each call sends the framework a message, telling it to increment the non-failing execution count of each constraint the program is using; hence, a logical time tick in a long-running program is effectively a "non-failing run". We found these calls trivial to insert, even into large and unfamiliar programs. For example, in our experiments with Apache and Memcached, we incremented logical time after 1,000 and 10,000 requests were processed, respectively.

6.1.3 Combined Avoidance Likelihood Model

The framework selects constraints by querying its *combined avoidance likelihood model*, which incorporates both the failure feedback model and the event pair model. The combined avoidance likelihood model is a probability distribution with an entry for each constraint. The value of an constraint's entry is the likelihood that it is effective as predicted by the event pair model, scaled by an exponential function of the constraint's observed failure rate, as recorded in the failure feedback model. Concretely, the amount of probability mass contributed by an constraint is:

$$P_{B,A} = \underbrace{(CI_{B,A} \times OI_{B,A} \times FC_{B,A})}_{\text{Event Pair Model}} \times \underbrace{e^{r_1 \times (1.0 - FR_{B,A}) - r_2} + s}_{\text{Failure Feedback Model}}$$

where r_1, r_2 are parameters of the exponential function used in the model and s is a smoothing factor that keeps the model defined and bounded by 0 and 1. We chose $r_1 = 8$, $r_2 = 0.7$, and $s = 0.001$. These choices cause the function to peak when $FR = 0$ and bottom out at 0.001 (s).

The intuition behind the combined model is the following. The event pair model is *predictive* – the model's features encode our inductive bias, and data (RPBs) refine the model's predictions. The failure feedback uses *feedback* to scale predictions made by the event pair model. The scaling factor varies exponentially with the constraint's failure rate – constraints that fail often are exponentially less likely to be drawn than ones that fail rarely or never. As failures and non-failing runs occur, Aviso refines its models. Over time, effective constraints' probabilities in the combined model grow and ineffective constraints' probabilities decrease.

Figure 8 illustrates the combined model. Failure correlation (FC) is computed based on RPBs from observed failures. Co-occurrence invariance (CI) and Order invariance (OI) are computed based on RPBs sampled from correct execution. The failure feedback model maintains failure rate values for each constraint, computed by monitoring constraint failures. Aviso uses the combined model to select constraints according to a probability function composed of the event pair and failure feedback models, as shown.

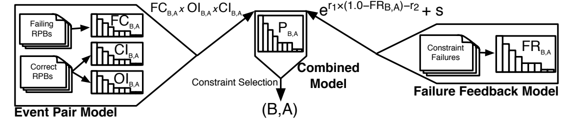


Figure 8. Aviso's statistical model. The event pair model tracks feature values for each constraint. The failure feedback model tracks constraints' failure rates. The combined model is comprised of the other two, yielding a selection probability for each constraint.

6.2 Distributing Constraints

All constraints start equally likely to be drawn. As program instances run, Aviso samples non-failing RPBs and stores them as the program runs. When the program fails, the framework generates constraints. The framework initializes the event pair model using the stored RPBs and the combined model assigns each constraint an initial probability based on the event pair model. The failure feedback model is ignored at this point because before any executions, all constraints' failure rates are undefined. Later, when a program instance starts, it queries Aviso for constraints. Aviso selects a constraint and sends it to the instance.

Aviso periodically instructs program instances to run with no constraints to establish the application's baseline failure rate. Initially, Aviso sends no constraint 10% of the time; the rate drops to 1% after seeing enough executions without any constraints to establish 95% statistical confidence that the observed baseline failure rate is within 5% of its actual value, assuming a binomial distribution of failures.

We continuously compute χ^2 statistics for each constraint to determine the significance of the difference of the constraint's observed failure rate and the baseline failure rate. We use a 2x2 contingency table and consider a difference to be significant if the χ^2 test indicates it to be with at least 95% probability. When a constraint that significantly decreases the failure rate is identified, Aviso uses that constraint 75% of the time. However, Aviso continues to draw constraints from the combined model for two reasons: (1) it is important to keep the baseline failure rate up to date; constraints may *lose* their significance if the baseline rate changes. (2) there may be other constraints with larger significant decreases

in failure rate; halting its exploration, Aviso may settle for a non-optimal constraint.

6.2.1 Handling Multiple Failures

Up to this point, our discussion has assumed that all failures can benefit from a common pool of constraints. However, programs are likely to have more than one type of failure, stemming from more than one bug. Aviso also deals with multiple failures. The key is to maintain a separate model for each *failure class*. A failure class is identified by the content of the RPB at the point of failure. When a failure occurs, the post-failure RPB is compared to the RPBs collected from failures in each failure class, using symmetric set difference. The RPB is assigned to the class to which it is most similar, unless it is not at least 80% similar to any class, in which case a new failure class is created.

When a program instance starts and queries the framework for constraints, Aviso selects a constraint from each failure class according to its own model. The program instance is then sent one constraint per class. The constraint-less failure rate information is shared across classes. On a failure or successful run, all classes' models are updated.

There may be two unrelated failures that occur with similar RPBs. If the failures are assigned to the same class, only one constraint will be applied to starting program instances. As a result, it is possible that only one failure or the other will be prevented. We accommodate this situation by allowing Aviso to split a failure class in two if no constraint significantly decreases the failure rate after a fixed number of experiments. The purpose of this process is to select two constraints for what was previously a single failure class and thereby prevent both failures.

7. System Implementation

We built a complete implementation of Aviso including the profiler, instrumenting compiler, runtime system, and the constraint selection and avoidance-sharing framework[1]. We implemented the sharing profiler using PIN [14]. Dominance pruning and event placement were implemented in LLVM [9]. The framework and runtime were implemented from scratch.

7.1 Framework Implementation

The framework was implemented in about 3000 lines of Go code. The statistical models and constraint generation were implemented from scratch in the framework.

The framework exposes a messaging API. The API provides calls for the runtime to query for constraints, to send RPBs for sampled, non-failing periods and after failures occur, and to send logical time ticks. The API works over the network, via HTTP. The framework and runtime-enabled program instances form a distributed system that implements Aviso. Using HTTP as the messaging protocol for the distributed system makes it flexible, portable, and suitable for use in cloud environments such as Google AppEngine or Amazon EC2. Furthermore, its simple interface lets the framework be trivially replicated and lets replicas be load balanced for further scalability. Replicas' statistical models could be kept consistent via consensus, or simply operate independently.

7.2 Runtime Implementation

We implemented the runtime in a library with an event handling API. Synchronization, signal and sharing events make calls to the API. When a thread makes an event call, it records the event with a timestamp in a thread-local queue. When the thread ends, the timestamped events are serialized to a file. Timestamps are collected at nanosecond resolution using `clock_gettime`'s `CLOCK_MONOTONIC` clock. We use thread-local event queues and timestamps to collect events because they are faster than an earlier version of our system, which used a serializing event queue shared across threads.

Constraints are shared object plugins to the runtime. Each exposes a factory method to instantiate its constraint. When the program starts up, the runtime receives constraint descriptions as text. The runtime uses a simple, custom, templated code generator to produce C++ code from the text. We then call out to gcc to compile the code to a shared library that is loaded by the program. Program instances cache compiled constraints, so code generation and compilation need only be performed once; subsequent executions that call for the same constraint can reuse the previously generated constraint plugin.

The runtime was built for concurrent performance. Its only shared data structure is the state associated with active constraints; everything else is thread-local. Accesses to the shared structure are rare: each thread has a thread-local list of events that may require an access to the shared structures. The common case is for a thread executing an event to check its list and continue without accessing the shared structure. Only when a thread hits an event involved in a constraint must it consult the shared structure. This arrangement minimizes the chance that Aviso's data-structures will lead to serialization and poor performance.

8. Evaluation

We evaluate Aviso along several dimensions. First, we show Aviso's efficacy in avoiding failures. Second, we show that Aviso's overheads are reasonable both during data collection and when actively avoiding failures. Third, we characterize Aviso's constraint selection process. Finally, we characterize the Aviso's dynamic behavior.

8.1 Experimental Setup

We evaluated Aviso using several buggy parallel programs.

Out-of-the-Box Benchmarks. Our main results are based on experiments with one cloud application, one server application, and one desktop application made to run with Aviso "out of the box".
Memcached-1.4.4 is an in-memory key value store with an atomicity violation that leads to data-structure corruption and lost updates under heavy update load. We added a single assertion that detects the data-structure corruption when a thread writes to a deallocated table cell and aborts execution. The data-structure invariant that our assertion checks is the cause of the lost updates, but the assertion is oblivious to the lost update problem; to write the assertion, a programmer would not need to understand the lost update failure. We manually added a single Aviso call to the server to send a logical time update every 10,000 requests. Inserting this call was trivial even without being familiar with the codebase. For profiling, we initialized a key-value store with 10 keys storing integers that 8 threads accessed. We used a mix of accesses that was 90% reads and 10% updates. For tests, we used a 10-key store and the same thread count and operation mix.
Apache-2.0.46 is a web server with atomicity violations in its in-memory cache subsystem that lead to crashes when concurrently servicing many php script requests. Our server setup is Apache with mod_php loaded, in-memory caching enabled, and serving the time of day via a php script. We added a single Aviso call to send a logical time update every 10,000 requests. As with Memcached, inserting the call was trivial. For profiling, we used `ApacheBench` to issue 1,000 requests from 8 concurrent request threads for a static `html` page, then 1,000 requests from 8 threads for our php time server. For tests, we let the server run continuously until a fail-

[1] Aviso will be available for download at the authors' website.

ure. We sent time-of-day requests in groups of 10,000. To vary the workload, each group of requests was sent by a number of threads uniformly randomly chosen to be between 2 and 8.

AGet-0.4 (Figure 1) is a download accelerator with an atomicity violation in its signal handler that leads to output corruption. To detect failures, we manually added an assertion that aborts when it detects output corruption. The assertion compares a count of bytes written to the downloaded file to the sum of per-thread byte counts. The symptom of the failure is that these counts are not equal. Note that adding this assertion did not require understanding how to *prevent* the failure. We needed to understand only that the number of written bytes reported by AGet should match the number of bytes in the output file. To profile AGet, we downloaded a 1MB file using 8 threads from a local network resource twice, once letting it complete and once interrupting it with `SIGINT`. To test AGet, we started downloading a 700MB Linux image and interrupted the download with a `SIGINT` after 1s.

Schedule-patched Benchmarks. To further demonstrate Aviso's applicability, we conducted experiments with two other desktop programs. Unlike our first three benchmarks, we altered these two programs to amplify their failure rate. We applied patches that use sleep statements to lead execution toward failing schedules, similar to prior work [5, 27, 28]. These schedule patches are *not* essential – Aviso could be applied without them; we used them to facilitate experiments. Despite the patches, these results show Aviso's effectiveness for several reasons. First, the program is unchanged except for a single call to `usleep`. Second, the increased failure rate does not affect constraint generation or selection, except to reduce the time required for both. Third, events involved in the failure are identical in the patched and non-patched versions.

Transmission-1.42 (Figure 7) is a bittorrent client with a use-before-initialize error that leads to an assertion failure. To profile Transmission, we downloaded a Linux iso torrent without the schedule-altering patch. We ran tests on Transmission by running with the schedule-altering patch applied and downloading a non-existent torrent, which triggers the failure, causing a crash.

PBZip2-0.9.1 is a compression utility with a use-after-free error that leads to a crash. To profile PBZip2, we ran it under our profiler and first compressed, then decompressed, a 10MB text file. We experimented with PBZip2 by compressing a 250MB file. Aviso diagnosed the failure by watching for crashes and failed assertions.

8.2 Bug Avoidance Efficacy

Our main finding is that Aviso made our benchmarks fail less frequently, as shown on the plots in Figure 9. The plots show *on a log scale* the number of failures observed in our experiments for Aviso and for the baseline without Aviso, as well as a theoretical worst case. The slope at a point on a curve is the instantaneous failure rate at that point.

For all benchmarks, Aviso's curve is lower than the baseline, indicating a decrease in the number of failures experienced. In Apache's case, Aviso decreased the number of failures exhibited in our experiments by *two orders of magnitude*. Memcached saw a decrease in failures of more than an order of magnitude. Other cases had less pronounced decreases, but still benefited from Aviso.

These data elucidate how Aviso searches for the most effective constraint. For the first few runs of the program, the number of failures for Aviso is commensurate with the number for the baseline. During these first runs, Aviso is building and refining its statistical model. After a few runs, Aviso's model guides it to an effective constraint. At this point, the slope of the curve becomes flatter than the baseline, *i.e.*, Aviso begins to consistently choose a constraint or constraints that avoid failures.

	Performance Overhead	
	Coll. Only	**Coll. & Avoid**
Transmission	8.8%	29.5%
AGet	26.2%	30.7%
PBZip2	2.6%	5.5%
Memcached	17.3%	16.7%
Apache	12.1%	15.9%

Table 1. Aviso's runtime overheads. These overheads are relative to baseline execution when collecting events only and when collecting events and avoiding failures.

8.3 Performance

Table 1 shows that Aviso's runtime overhead is low. Column 2 is the overhead of event monitoring only. The overhead ranges from less than 3% for PBZip2 to 26.2% for AGet, with an average overhead of 13.4%. These results show that when Aviso is only collecting information, the performance overhead is tolerable.

Collection and Avoidance Overhead. Column 4 shows the combined overhead of event monitoring and avoidance. Avoiding failures does not prohibitively increase Aviso's overhead. For Memcached, the overhead is about the same as that of event collection alone; for Transmission, our worst case increase, the overhead is 20.7% greater than the overhead of event collection.

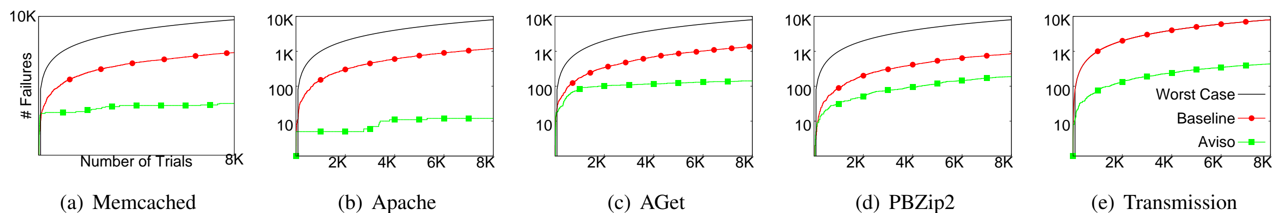
PBZip2 has very low overhead for both event collection and avoidance because threads spend a majority of the execution in a compression routine in `libbz2`. Avoidance adds little to the overhead because the most effective constraint for PBZip2 involves events that execute during shutdown; constraint activation checks and delays need only occur during shutdown, so they do not impede the execution.

AGet's event collection overhead is high relative to our other benchmarks because a majority of the program's execution is in a tight loop that includes two event calls. AGet's avoidance overhead is only slightly higher than its collection overhead because the events involved in the constraints that Aviso found effective execute only during signal handling. The increased overhead is due to an increase in constraint activation checks, not delays.

Memcached's overhead is nearly the same for both collection and avoidance: the constraints that Aviso found effective are not activated in the program's common case. The events involved in effective constraints execute only when the number of digits in the number stored in one of Memcached's table cells increases, which occurs rarely.

The key finding, then, is that when collecting events only, Aviso imposes a low performance penalty. When avoiding failures, the overhead is only slightly higher.

Contrasting Improved Reliability with Aviso's Overhead The data show that Aviso's overhead is non-negligible. These overheads are acceptable for two main reasons. First, the increase in reliability comes immediately and without the need for the programmer to understand how to fix the program. Patches are hard to write correctly, and hand-written patches may introduce bugs or degrade performance. For example, Memcached developers left the failure we studied unpatched for nearly a year after its initial report. They cited a 7% performance "regression" as one roadblock to committing a patch [1]. Aviso imposes a roughly similar performance overhead (16.7%) to the manually crafted solution and decreases the rate at which the failure occurs by nearly two orders of magnitude. Aviso does not require the programmer to understand how to fix the bug, let alone correctly patch the code to fix it. Furthermore, because Aviso operates automatically the gap between the first failure and Aviso's failure avoidance is a few minutes rather than the year required for the manual patch.



(a) Memcached (b) Apache (c) AGet (d) PBZip2 (e) Transmission

Figure 9. Aviso's improvement in reliability. We show data for (a)Memcached, (b)Apache, (c)AGet, (d)PBZip2, and (e)Transmission. The x-axis shows execution time in number of trials – logical time ticks for servers, executions for standalone applications. We ran each program for 8000 trials. The y-axis shows the the number of failures that have occurred at a given point in time *on a log scale*. The top (black) curve shows the worst case: every execution is a failure. The middle (red) curve shows the reliability of the baseline, compiled and run completely without Aviso. The bottom (green) curve shows the reliability with Aviso.

Second, Aviso's performance overhead saves programs from the potentially severe costs associated with failures. For example, Memcached's failure is a lost update that permits the store to be periodically corrupted but to continue executing. In safety-critical applications, data corruption is likely worse than performance degradation. Aviso provides the option of avoiding Memcached's data corruption at the cost of a modest performance hit.

8.4 Constraint Generation and Selection

Figure 10 characterizes how Aviso generates and selects constraints. Figure 10(a) shows that Aviso needs to experiment with only a small fraction of the constraints it generates to find effective constraints. Notice that the lower portion of the bars is considerably smaller than the upper portion: Aviso made only a small fraction of constraints available to program instances. Aviso effectively avoids failures, so this result shows that it selects effective constraints without having to observe many program instances.

Figure 10(b) shows that the number of constraints that Aviso makes available to program instances is small and for most of our benchmarks, most of the execution time was spent using a single effective constraint. These data reinforce the findings from Figure 10(a), *i.e.*, Aviso finds effective constraints after selecting and distributing only a few using its statistical model.

For Apache, Memcached, and AGet, the constraint represented by the bottom bar segment was used by Aviso for 92-99.7% of the execution time during our tests. For PBZip2, the bottom two bar segments account for nearly 80% of execution time; Aviso chose between these two constraints a majority of the time. These frequently chosen bottom segments all represent constraints that led to a statistically significant decrease in failure rates.

In concert with Figure 10(a), this result illustrates how Aviso works: Aviso initially selects effective constraints without having to experiment with them or directly observe their impact on failure rates. It chooses good constraints without experimenting by using its predictive event pair model. After initially selecting a constraint that turns out to be effective (*i.e.*, the event pair model's prediction was a good one), the failure feedback model biases Aviso to select the same constraint again. The data directly show this phenomenon. For example, in Apache's case, Aviso selected 16 different constraints, experimented with each, and observed their impact on the failure rate. The 16th turned out to be effective, preventing nearly all future occurrences of the failure. Due to the constraint's success, the failure feedback model ensured it was subsequently selected.

Aviso selects and tests constraints differently for Transmission than for the other benchmarks. Transmission's lower 14 bar segments provided a significant decrease in the failure rate. Aviso used one of these 14 constraints for about 85% of execution time.

The data in Figure 10(c) explain why Transmission is different and help further characterize Aviso's event pair model. The bar height shows the ratio of the total number of constraints generated during our experiments to the total number of pairs in the event pair model that were observed in sampled correct RPBs. We call this ratio the *coverage* of the model. If the event model has fewer pairs than constraints – *i.e.*, has low coverage – it is likely to predict effective constraints poorly. If the model has more pairs, it is more likely to be useful in assigning meaningful selection probabilities to more constraints. Note that coverage may exceed 1.0 if pairs in the model never show up in a post-failure RPB.

Transmission's model coverage is zero. The structure of Transmission's failure explains why: the failure occurs very early in the program's execution. The event pair model is primarily built from RPBs sampled from portions of correct execution. Transmission crashes early, so no RPBs are sampled, and the event pair model is of little use. Transmission's zero model coverage explains why Aviso was forced to experiment with more different constraints. Instead of predicting effective constraints, Aviso relied on the results of its experiments – the failure feedback model – to determine which constraints worked best.

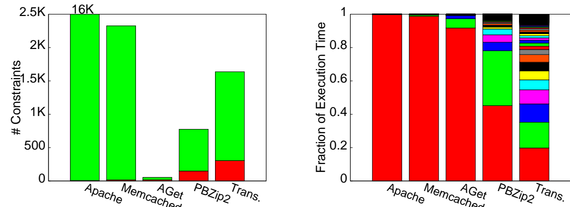
Our other benchmarks had event pair models with higher coverage. Apache's model contained nearly the same number of pairs as there were constraints. Memcached and AGet also had models with high coverage. Looking back to Figures 10(a) and (b), the impact of higher model coverage is clear. For benchmarks with higher model coverage, the fraction of constraints used is lower and the fraction of execution time spent using a small number of effective constraints is higher.

In summary, Figure 10 shows that when Aviso's statistical model has observed enough correct execution behavior, it makes good predictions, and Aviso is effective. If the model has low coverage, Aviso still selects effective constraints using its failure-feedback model.

8.5 Characterizing Dynamic Behavior

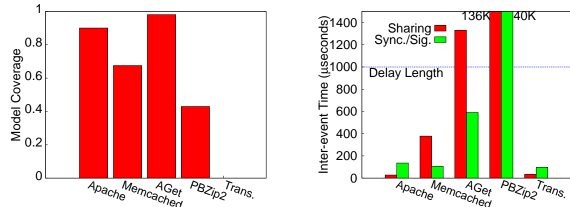
Using an instrumented version of the Aviso runtime, we characterized its dynamic behavior. For these experiments, we fixed a few runtime parameters: we chose the constraint used most frequently by Aviso during our main experiments, and we used a single fixed-size input. For PBZip2, we used the 250MB input file. For AGet, we downloaded a Linux image and interrupted execution (without crashing). For Apache, we issued 1M requests from 8 concurrent request threads. For Transmission, we downloaded a Linux image torrent, while for Memcached, we ran 80,000 client requests.

Figure 10(d) plots the rolling average time between events in μs during our experiments. These data justify Aviso's delay length. Recall that to avoid failures, events must be reordered by delays. In order to reorder events, events must be delayed long enough to allow an event in another thread to execute. The data show that the delay time is longer than the average inter-event time of



(a) **Characterizing constraint generation.** The bar height represents the total number of constraints generated after any failure that occurred during our experiments. The lower, red segment shows the number of constraints that were actually made available to a program instance by Aviso.

(b) **Characterizing constraint selection.** Each bar corresponds to a different benchmark. Each segment corresponds to a different constraint. The height of a segment represents the fraction of execution time during vetting that ran with that segment's constraint. The total number of segments in a bar is the number of constraints Aviso experimented with.



(c) **Characterizing of Model Coverage.** Bar height represents model coverage: the ratio of the total number of constraints generated to the total number of pairs in the event pair model that were observed in sampled correct RPBs.

(d) **Characterizing inter-event time.** Bars show rolling average time between events in μs. The left (red) bar reflects sharing events and the right (green) bar, synchronization and signal events. Longer intervals mean events are less frequent. The horizontal line shows Aviso's delay length.

Figure 10. Characterizing Aviso's behavior.

most benchmarks. In the average case, a delay will reorder some events. Despite PBZip2's large average inter-event time, Aviso's delay length is adequate because the length of the interval between events involved in the failure is less than the delay length.

The data in Figure 10(d) also help explain Aviso's overheads. For PBZip2, the average time between events is several orders of magnitude larger than for other applications. The length of the interval makes sense because PBZip2 spends most of its time in a compression library call. It is likely that PBZip2's long time between events contributes to the low overhead reported in Table 1. Each of the other benchmarks has a shorter interval between events than PBZip2. Correspondingly, the event collection overheads in Table 1 for the other benchmarks are slightly higher than PBZip2's.

AGet, with the highest performance overhead, has several events on its inner loop. We expected these events to result in a short interval between events, explaining its overhead. However, as Figure 10(d) shows, AGet's inter-event interval was moderately *longer* than most other cases. Table 2 shows data that explain AGet's performance and further characterizes Aviso.

The data in the table illustrate several different sources of overhead: excessively frequent events, leading to many discarded events; constraint activation checks; and delays due to constraints. AGet's overhead is likely to come from frequent constraint activation checks. In AGet, 1 out of every 3 events requires the extra computation of a constraint activation check. In contrast, 0.4% of Apache's and 7.25% of Memcached's events required such checks. Constraint activation checks require holding a lock and accessing shared state, so they are more costly than those that do not.

	Sharing Events		**Sync/Sig Events**		**Constraints**		
	# Evts	# Discard	# Evts	# Discard	# Chks	# Strt	# Delays
Apache	44.6M	37.9M	1.5M	34K	27K	65	56
Memcached	200K	13K	1.2M	204K	87K	16	87
AGet	46K	0	92K	2	46K	5	40
PBZip2	227	0	1042	4	81	8	8
Transmission	10.6M	5.0M	2.0M	5	96	1	1

Table 2. Aviso's dynamic behavior. Columns 2 and 3 show the total number of sharing events and the number of sharing events discarded due to online pruning. Columns 4 and 5 show the total number and number discarded of synchronization and signaling events. Column 5 shows the number of times an event in an available constraint executes, requiring a check to see if the event activates the constraint. Column 6 shows the number of times a check actually leads to a constraint's instantiation. Column 7 shows the number of times an event is delayed by a constraint.

The data in Table 2 show that a large fraction of Apache's events (over 80%) were discarded due to online pruning. The high rate of discards suggests that events are frequent; the very short inter-event time shown in Figure 10(d) corroborates this fact. Intuitively, such high-frequency events seem like a performance problem; however, Apache's event frequency did not impose excessive overhead – around 15%. Most of Apache's events did not require activation checks, instantiations, or delays. As a result, its events were inexpensive, requiring just a few access to thread-local memory. The absence of complex computation or serialization on global state is likely the reason for Apache's low overhead.

Delays were very infrequent across all our benchmarks, occurring mostly in uncommon case code. In PBZip2 8 worker threads delayed a cleanup thread. In Transmission, a delay during startup code prevented a use-before-initialization error. In Apache, a delay during a request cache flush prevented a crash. In AGet, a delay during signal handling prevented a crash. In Memcached, a delay during a rare-case update prevented data corruption.

To summarize, delays were not a problem in our tests because they were infrequent and in rare-path code. Event frequency alone did not dictate performance, although having very infrequent events seemed to lead to lower overhead (*e.g.*, PBZip2). Constraint activation checks seemed to be a more costly source of overhead than we expected, especially when events were frequent (*e.g.*, AGet).

9. 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.

Loom [26] is a system for patching concurrency bugs in running programs. Loom is like Aviso in that it aims to prevent failures between when a failure occurs, 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 constraint. 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 avoiding atomicity violations [4, 12, 13, 21, 22]. Isolator [21] and ToleRace [22] prevent single-variable atomicity violations, but do not handle the broader class of failures addressed by Aviso. AFix [4] 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. Atom-Aid [13] and ColorSafe [12] address single- and multi-variable atomicity bugs. These systems are unlike Aviso in that they only handle atomicity bugs, and need special hardware support. Other systems have proposed using hardware support to force executions to adhere to tested schedules [27, 28]. 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 avoids failures even in untested code. Dimmunix [7] and Communix [6] provide automated deadlock immunity for Java programs. Like Aviso these systems identify and avoid failures mostly automatically, and Communix systems share failure avoidance capability. These systems are limited in that they only address deadlocks.

Determinism [2, 15, 17] enforces one event interleaving in every execution. Aviso also affects event orderings. Aviso differs in that it does not restrict the interleaving of the entire program. Instead, all possible executions are permitted, except where restricted by constraints.

Avoiding Other Failures There is other work on failure avoidance not specifically for concurrency. Rx [20] uses checkpointing to recover from failures. Aviso differs from Rx in that it avoids failures, rather than just failing and recovering, and Aviso does not require checkpointing support. Failure-oblivious computing [23] permits execution to continue after a failure. This technique works well for some non-critical errors, but is otherwise of limited use.

Exterminator [16] was among the first to explore what it called "collaborative bug correction", inspiring Aviso's cooperative approach to failure avoidance. Unlike Aviso, Exterminator focuses on avoiding failures due memory errors (*e.g.*, buffer overflows, etc.), not concurrency errors, and it does so with just slightly higher overheads than Aviso (25% average runtime overhead).

ClearView [19] identifies invariants, and when they are violated produces a patch preventing future violations. ClearView is similar to Aviso in that it provides avoidance by avoiding behavior observed in failing runs. ClearView and Aviso both monitor their impact to determine the most effective strategy (constraints in Aviso, and patches in ClearView). Aviso differs in that it does not use invariants, and focuses on concurrency.

10. Conclusions

We presented Aviso, a system that automatically avoids concurrency-related program failures by empirically determining fault-free execution schedules. Aviso leverages a community of instances of a program and uses statistical techniques to quickly determine which program events (and their order) are the culprit of a failure. We built Aviso in software only (not relying on special hardware), and our evaluation showed that Aviso increases reliability significantly (orders of magnitude reduction in failure rate in some cases) and leads to overheads acceptable in real production runs. As our future work, we will explore expanding our statistical model to incorporate quality of service criteria, and using Aviso to implement speculative, empirically tuned performance optimizations.

Acknowledgments

We thank the members of the UW Sampa group and the anonymous reviewers for their feedback and help. We also thank Tony Fader and Adrian Sampson for their early contributions. This work was supported in part by the National Science Foundation under grants CCF-1064497 and CCF-1016495, and gifts from Microsoft.

References

[1] Issue 127: incr/decr operations are not thread safe. http://code.google.com/p/memcached/issues/detail?id=127.

[2] T. Bergan et al. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[3] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[4] G. Jin et al. Automatic atomicity-violation fixing. In *PLDI*, 2011.

[5] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

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

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

[8] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *ASPLOS '12*, 2012.

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

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

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

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

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

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

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

[16] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI '07*, 2007.

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

[18] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, 2009.

[19] J. H. Perkins et al. Automatically patching errors in deployed software. In *SOSP*, 2009.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP*, 2005.

[21] S. Rajamani et al. Isolator: Dynamically ensuring isolation in concurrent programs. In *MICRO*, 2009.

[22] P. Ratanaworabhan et al. Detecting and tolerating asymmetric races. In *IEEE Transactions on Computers*, 2011.

[23] M. Rinard et al. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[24] Y. Shi et al. Do I use the wrong definition? defuse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[25] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. In *ASPLOS 2012*, 2012.

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

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

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