

Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation*

Guoliang Jin Aditya Thakur Ben Liblit Shan Lu

Computer Sciences Department
University of Wisconsin–Madison, Madison, WI, USA
{aliang,adi,liblit,shanlu}@cs.wisc.edu

Abstract

Fixing concurrency bugs (or *crugs*) is critical in modern software systems. Static analyses to find crugs such as data races and atomicity violations scale poorly, while dynamic approaches incur high run-time overheads. Crugs manifest only under specific execution interleavings that may not arise during in-house testing, thereby demanding a lightweight program monitoring technique that can be used post-deployment.

We present Cooperative Crug Isolation (CCI), a low-overhead instrumentation framework to diagnose production-run failures caused by crugs. CCI tracks specific thread interleavings at run-time, and uses statistical models to identify strong failure predictors among these. We offer a varied suite of predicates that represent different trade-offs between complexity and fault isolation capability. We also develop variant random sampling strategies that suit different types of predicates and help keep the run-time overhead low. Experiments with 9 real-world bugs in 6 non-trivial C applications show that these schemes span a wide spectrum of performance and diagnosis capabilities, each suitable for different usage scenarios.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—parallel programming; D.2.4 [Software Engineering]: Software/Program Verification—statistical methods; D.2.5 [Software Engineering]: Testing and Debugging—debugging aids

*Supported in part by AFOSR grants FA9550-07-1-0210 and FA9550-09-1-0279; DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0621487, CCF-0701957, CCF-0953478, and CNS-0720565; and a Claire Boothe Luce faculty fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

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.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10...\$10.00

General Terms Experimentation, Reliability

Keywords concurrency, statistical debugging, random sampling, bug isolation

1. Introduction

1.1 Motivation

Concurrency bugs (or *crugs*), such as data races [9, 12, 36] and atomicity violations [13, 26], are among the most troublesome software bugs. The unique non-determinism of crugs makes them difficult to expose during in-house testing. As a result, many crugs slip into production runs and manifest at user sites. Even worse, many crugs can cause severe software failures, varying from data corruption to program crashes [16]. Crugs have caused real-world disasters in the past, such as the Northeastern Blackout of 2003 [37]. Growing use of concurrent programs on multi-core hardware means that software reliability is increasingly threatened by crugs. Tools for diagnosing production-run failures in concurrent software are sorely needed.

To date, it has been extremely difficult to diagnose production-run software failures caused by crugs. Performance is the biggest challenge. Most prior crug detection tools either have huge overhead (10×–100× slowdown [36]) or require specialized hardware that does not yet exist [26]. Accuracy is another challenge, as many previous detectors [13, 36] have high false positive rates. For example, Narayanasamy et al. [31] show that only about 10% of real data races are harmful and could cause software failures. Recently Marino et al. [29] and Bond et al. [5] smartly use sampling to improve the performance of race detection. Although inspiring as a race detector, it unavoidably suffers accuracy and coverage problems in failure diagnosis; we discuss this further in Section 5.4 and Section 6. Furthermore, it can be very hard for developers to reproduce field-detected concurrent software failures which manifest only under special interleavings. Even if the bug-triggering input is known, it may still take developers several days to reproduce a crug [32]. As a result, failure diagnosis is a nightmare for the developers of concurrent programs.

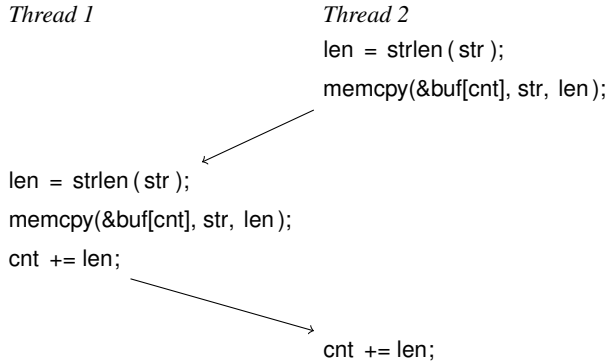


Figure 1: Atomicity violation bug from the Apache HTTP Server. The variables `buf` and `cnt` are both shared. Both threads are inside function `ap_buffered_log_writer`.

The Cooperative Bug Isolation project (*CBI*) aims to automatically diagnose production-run software failures with small run-time overhead [8, 23, 25]. It achieves this goal through three steps. First, it statically instruments a program at particular program points so as to monitor various predicates on program state and behavior, such as variable value predicates (e.g., $x > y$) or the paths followed at conditional branches. Next, at run-time, it gathers feedback about program execution by collecting these predicate samples, as well as corresponding labels of execution results (success or failure). Lastly, *CBI* performs *statistical debugging*: statistical analysis of aggregated feedback data to identify program (mis)behaviors that are highly correlated with failure. The *CBI* framework achieves low monitoring overhead through sparse random sampling of the instrumentation and by collecting information from many user sites.

Unfortunately, prior *CBI* work is poorly suited to diagnosing software failures caused by crugs. The root causes of concurrency bugs fundamentally involve sequences of actions from *multiple* threads. They cannot be captured by predicates used in prior *CBI* work, which focus on *one* thread at a time.

Figure 1 shows an example simplified from a real-world crug in the Apache HTTP Server [1]. Shared variable `cnt` is the tail index of shared buffer `buf`. Every thread appends log messages to this buffer based on the index. Unfortunately, without proper synchronization, buffer updates and index accesses from different threads can race with each other and lead to garbage data in the log, as shown in Figure 1.

Previous *CBI* tools fail to diagnose this problem. In our experiment, none of the standard predicates behaves differently in failing versus successful runs with at least 95% confidence (*CBI*'s standard acceptance threshold to counteract the effects of sampling noise). This is because the software's misbehavior (garbage log data) can happen with normal variable values (e.g., `cnt` remains in bound) and normal execution paths, if considered only within individual threads. Thus we require new instrumentation schemes in order to diagnose software failures due to crugs.

1.2 Contributions

This paper presents Cooperative Crug Isolation (*CCI*), a low-overhead dynamic strategy for diagnosing production-run failures in concurrent programs. Following the Cooperative Bug Isolation philosophy, *CCI* monitors interleaving-related predicates at run time; leverages sampling to keep the run-time overhead low; and uses statistical models to process run-time information aggregated through many runs and many users and identify the root causes of production-run failures.

Applying the Cooperative Bug Isolation idea to crugs raises two major questions:

What types of predicates are suitable for crug diagnosis? Instrumentation must balance failure-predictive power and computational simplicity. Poorly-designed predicates may be unable to explain any crug failures. Yet costly predicates must use low sampling rates in order to provide performance guarantees, thereby delaying diagnosis. If the evaluation of a predicate relies on long and continuous monitoring, it may be unsuitable for sampling.

How can we sample predicates that are related to concurrency bugs? Previous *CBI* work made predicate sampling decisions independently in each thread at each execution point. *CCI* sampling is much more complex. It may require cross-thread coordination, because crugs involve multiple threads. It must also keep each sampling period active for some time, because crugs always involve multiple memory accesses. Proper sampling design affects both the number of predicates that can be collected as well as the correctness of the collected data.

There may be no single solution to all of the above challenges. We consider three different types of predicates together with new sampling strategies that support each:

1. **CCI-Havoc** tracks whether the value of a memory location is changed between two consecutive accesses from one thread. This captures the change of program states in the view of *one* thread at *two* nearby points. *CCI-Havoc* monitoring is supported by thread-independent and bursty-style sampling in *CCI*.
2. **CCI-FunRe** tracks function re-entrance: simultaneous execution by multiple threads. This captures the interaction among *multiple* threads at a *coarse* granularity. It is supported by thread-coordinated and unconditional sampling.
3. **CCI-Prev** tracks whether two consecutive accesses to one memory location come from the same thread or distinct threads. This captures interactions among *multiple* threads at a *fine* granularity. It is supported by thread-coordinated and bursty sampling.

These three schemes offer different types of information that may help diagnose crug failures. They provide different trade-offs between performance and failure-predicting

capability, and demonstrate different ways of sampling in concurrent programs.

Specifically, this paper makes the following contributions:

- A new suite of predicates that effectively diagnose production-run failures in concurrent programs. Each reflects a common type of root cause for synchronization failure. Together, they span a wide spectrum of trade-offs between diagnostic capability and complexity.
- A new suite of sampling schemes that support different types of crug instrumentation. Interleaving-related predicates are more complex than those used in previous CBI work. CCI offers suitable sampling strategies to support proposed and future interleaving-related predicates: thread-independent and thread-coordinated sampling; bursty and non-bursty sampling; and conditional and un-conditional sampling.
- A tool, CCI, that diagnoses production-run failures in concurrent programs. CCI provides:
 - **Low run-time overhead** benefiting from the sampling techniques developed herein.
 - **Low false positive rate** benefiting from statistical debugging. Many previous detection tools [12, 13, 36] have high false positive rates, because races and atomicity violations can be benign. As a failure diagnosis tool, CCI is immune to these false positives, because its statistical analysis leverages information about whether a particular run succeeded or failed and identifies those predicates whose values are truly correlated with observed failure.
 - **Good diagnosis coverage** benefiting from the crug-related predicates used in CCI. CCI-Prev predicates capture data races and atomicity violations; CCI-FunRe predicates capture misuse of thread-unsafe functions; and CCI-Havoc predicates capture atomicity violations. These are among the most common causes of crugs [27].

We validate CCI by using it to identify root causes of real-world failures in several concurrent C applications, including Apache [1], Cherokee, Mozilla, PBZIP2 [15], and the SPLASH-2 [44] benchmarks.

Experimental results show that CCI is very effective, dramatically outstripping CBI, for crug diagnosis. Traditional CBI tools fail completely, providing no predictors for any of our buggy concurrent test subjects. The predictors reported by CCI, however, accurately point to the root causes of a wide variety of failures. Furthermore, CCI achieves this excellent failure diagnosis with small run-time overhead (mostly within 10%), thanks to its sampling mechanisms.

The remainder of this paper is organized as follows. Section 2 provides an overview of the CBI framework. Sections 3 and 4 respectively describe our instrumentation predicates and sampling strategies in detail. We present experimental

results in Section 5 and discuss related work in Section 6. Section 7 concludes and suggests directions for future work.

2. Background

CCI builds upon Cooperative Bug Isolation (CBI), a framework for lightweight instrumentation and statistically-guided debugging [23, 25]. CBI collects information about program execution from both successful and failing runs and applies statistical techniques to identify the likely causes of software failures.

2.1 Data Collection Using Sampled Instrumentation

CBI’s instrumenting compiler uses source-to-source transformation to add instrumentation code that monitors the values of predicates at particular program points, called *instrumentation sites*. In this paper, we denote recording the value of predicate p at instrumentation site s as $\text{record}(s, p)$. The traditional CBI framework tracks following types of predicates:

Branches: Each branch is an instrumentation site. Two predicates are associated with each site: one is true when the true branch is taken, and the other is true when the false branch is taken.

Returns: Each function return point is an instrumentation site. A set of three predicates at each site track whether the returned value is negative, zero, or positive.

Scalar-pairs: At each assignment to a scalar variable x , one instrumentation site is created for each other same-type in-scope variable y . Each such site has three predicates, recording whether x is smaller than, larger than, or equal to y . Value comparisons between x and program constants are also added, one instrumentation site per constant.

During execution, the instrumentation code collects predicate profiles recording whether each monitored predicate was ever observed, and if observed, whether it was ever true. The feedback report from each run is a bit vector with two bits for each predicate (observed and true), plus one final bit indicating overall execution success or failure.

2.2 Data Analysis Using Statistical Debugging

CBI’s statistical debugging models operate on large collections of feedback reports. The models assign a score to every available predicate and identify the best failure predictor among them. Intuitively, a good predictor should be both *sensitive* (accounts for many failed runs) and *specific* (does not mis-predict failure in successful runs). Thus, a good predictor is an instrumented predicate that is true in many failed runs but very few successful runs. CBI’s scoring model considers both sensitivity and specificity to select top predictors.

Prior work shows that the best predictor often points to a bug that is responsible for many observed failures. An iterative ranking and elimination process continues to pick up the best remaining predicate to explain the remaining failures until all failures are explained or all available predicates are

discarded. We omit a detailed discussion of these statistical models here, as they are identical to those used in prior CBI work. The main focus of the present research is how to collect informative raw data efficiently in the first place.

Monitoring overheads must be very low for this approach to be feasible in post-deployment environments. The CBI framework achieves this goal through sparse random sampling. At run time, each time an instrumentation site is reached, a Poisson (memory-less) random choice decides whether or not the predicate information associated with that site will be collected. In this paper, “[*instr*;]?” represents the random sampling of the instrumentation *instr*. Sparse sampling means that most instrumentation code is not run, and therefore most run-time events are not actually observed. However, sampling is statistically fair, so the small amount of data that is collected is an unbiased representation of the complete-but-unseen data. Therefore, given a large number of user runs and appropriate statistical models, the root causes of failure emerge as consistent signals through the sparsely-sampled noise.

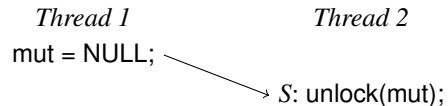
2.3 Practical Suitability for Concurrency Bugs

Prior CBI work has not focused directly on concurrency-related issues. There are good reasons to believe, however, that the two can be a sensible, natural fit in practice.

Analysis of feedback reports does not depend on the exact failure location (e.g., stack trace). The statistical models can diagnose both fail-stop errors as well as nonfatal bugs which allow execution to continue. They only require outcome labels marking each run as successful or failed. If one can only recognize crashes as failures, then we can diagnose crugs that lead to crashes. If one can automatically recognize corrupted output, then we can diagnose crugs that corrupt output. If users must manually flag runs as failed, then only crugs that sufficiently disturb users will be diagnosed.

Thus, statistical debugging is “pay as you go”: developers receive diagnoses of whatever bugs they can recognize when they arise. This was true in prior CBI work and it remains true in CCI. Conversely, the statistical models automatically ignore benign race conditions that never cause noticeable failures. This is a key benefit of our approach.

The sampled nature of CBI and CCI instrumentation might be seen as a limitation: if crugs are rare, then sampling could miss key clues. However, we see this as a strength. Even if crugs manifest less frequently than sequential bugs, CCI can help diagnose them in the field as long as the bugs have caused a sufficient number of failures just like CBI diagnoses sequential bug failures. For deployed software, diagnosis preference is usually given to those bugs (whether sequential or concurrent) that have manifested and bothered a sufficient number of users. Statistical debugging excels in exactly this scenario. Indeed, it would be quite reasonable to deploy a program with both CBI and CCI instrumentation, then let bugs “compete” for developer attention via the single statistical model that CBI and CCI both share.



(a) Failing run: $local_S = \text{false}$, $remote_S = \text{true}$



(b) Correct run: $local_S = \text{true}$, $remote_S = \text{false}$

Figure 2: State of CCI predicates in two different thread interleavings. Above code is simplified from a data race bug from PBZIP2 in which thread 1 nullifies the shared mutex variable, *mut*, when thread 2 is still using *mut*.

3. CCI Instrumentation Schemes

CCI does not gather anything approaching a complete trace at run time. Rather, it collects just a small amount of potentially-informative data that is readily available with minimal overhead. The format of such information (i.e., predicates) must be designed carefully, taking the following factors into account:

Simplicity. Simpler predicates allow more intensive sampling without causing excessive slowdown during production runs.

Failure-predictive capability. Predicates must reflect common root causes of crugs in order to explain crug failures during production runs.

Suitability for sampling. Sampled feedback data is incomplete. This limitation can make some types of predicates hard to evaluate. We discuss this further in Section 4.3.

It is unlikely that a single instrumentation strategy can satisfy all of these requirements. Therefore, in CCI, we design and explore three types of predicates that represent different design trade-offs: CCI-Prev, CCI-FunRe, and CCI-Havoc. The remainder of this section discusses the design of these three instrumentation schemes and how to monitor these predicates at run time without concern for sampling. The next section develops sampling strategies suitable for use with these three instrumentation schemes.

3.1 CCI-Prev Scheme

CCI-Prev tracks whether two successive accesses to a given location were by two distinct threads or were by the same thread both times.

3.1.1 CCI-Prev Instrumentation Sites

CCI-Prev monitors each instruction *I* that might access a shared location *g*. Each such instruction can exhibit two possible behaviors at run time: either the thread now accessing *g* at *I* was the same thread that accessed *g* previously, or

```

1 lock(glock);
2 changed = test_and_insert(&g, curTid);
3 record(s, changed);
4 access(g);
5 unlock(glock);

```

Figure 3: CCI-Prev instrumentation. The original program code, represented here as `access(g)` on line 4, was either a read or a write of possibly-shared memory location `g`.

the previous access was by a different thread. In CBI terms, we say that I constitutes a single instrumentation site with two predicates: $local_I$ is true if the previous access was from the same thread, while $remote_I$ is true if the previous access was from a different thread. Each time instruction I is executed, exactly one of these two predicates must be true, and the other false. For example, when execution follows Figure 2a, CCI records that $remote_S$ is true. Conversely, when the interleaving follows Figure 2b, $local_S$ is true.

3.1.2 Diagnostic Potential of CCI-Prev

CCI-Prev predicates are closely tied to the root causes of crugs, such as atomicity violations and data races. Most atomicity violation bugs happen when one thread’s consecutive accesses to a shared variable are non-serializably interleaved with accesses from a different thread [26, 42]. The Apache bug shown in Figure 1 is one such example: two consecutive read accesses to `cnt` in thread 2 are interleaved. Our $remote_I$ predicates capture these bad interleavings. Data races occur when conflicting accesses from different threads touch the same shared variable without proper synchronization. The PBZIP2 bug shown in Figure 2 is a typical example of a data race bug, and likewise can be recognized using CCI-Prev predicates.

CCI-Prev predicates are definitely not true atomicity-violation or race detectors. Extra information, including memory access type (i.e., read vs. write) and synchronization (i.e., when and which locks are acquired and released), would be needed to precisely record atomicity violations and data races. We intentionally ignore this information to keep instrumentation simple, following the CCI design principles mentioned above.

3.1.3 CCI-Prev Instrumentation Strategy

CCI-Prev instrumentation requires recording which thread issued the latest access to each shared memory location at run-time.

Figure 3 illustrates the instrumentation inserted by CCI at compile time. This instrumentation interacts with a global last-access hash table that stores mappings from memory addresses to the ID of the thread that last accessed each memory location. This hash table is checked and updated at every instrumentation point by function `test_and_insert` on line 2. As a result of `test_and_insert`, the local variable `changed` is set to true if the entry in the hash table corresponding to `&g` does

not equal `curTid` (i.e., the ID of the current thread), and to false otherwise. In addition, `test_and_insert` inserts the entry `&g ↦ curTid` into the last-access hash table. Subsequently, the value of `changed` is used to record CCI-Prev predicates as shown on line 3. Also note that in this scheme the variable `glock` is a global lock that is used to synchronize accesses to the entire hash table and ensure that the instrumented code appears to execute atomically. Notice that this global lock will never cause previously impossible program interleavings or deadlocks. Future work can further optimize CCI-Prev using lock-free hash tables [18] or fine-grained locks.

One practical challenge in our static instrumentation is that a single code statement might access multiple possibly-shared locations, and thereby require multiple hash table inserts and look-ups. We solve this by automatically rewriting complex source statements as multiple, equivalent, simpler statements which each accesses at most one possibly-shared location.

Our current implementation uses an extremely lightweight analysis to decide which accesses touch possibly-shared memory locations. We do not instrument accesses to const-qualified data. We instrument every direct access to a named global variable that is not declared as thread-local or a named local variable whose address is taken anywhere in the containing function. Finally, we instrument every indirect access through a pointer. Future work can use escape analysis to prune out unnecessary instrumentation sites and further improve CCI’s performance.

3.2 CCI-FunRe Scheme

CCI-FunRe tracks whether the execution of a function F overlaps with the execution of F from a different thread.

3.2.1 CCI-FunRe Instrumentation Sites

CCI-FunRe monitors each function F . Each execution of F can exhibit two possible behaviors at run time: either no other thread executes F during the current thread’s execution of F , or there exists another thread that executes F simultaneously. We say that F yields a single instrumentation site with two predicates: $Reent_F$ is true if the execution of F overlaps with the execution of F from another thread, while $NonReent_F$ is true if there is no such overlap. Each time function F is executed, exactly one of these two predicates must be true, and the other false. For example, $Reent_{ap_buffered_log_writer}$ is true for the interleaving shown in Figure 1.

3.2.2 Diagnostic Potential of CCI-FunRe

CCI-FunRe captures thread-interaction at function-level granularity. Since this granularity is much coarser than the access-level granularity adopted by CCI-Prev, CCI-FunRe has lower overhead than CCI-Prev under the same sampling rate.

CCI-FunRe covers a common pattern of crugs: a function is not thread-safe but is invoked by multiple threads concurrently. The Apache bug shown in Figure 1 is one such example. Function `ap_buffered_log_writer` updates the global log of Apache server. It should never be executed by mul-

```

1 F (...) {
2   if ((local_FCount++)==0)
3     oldFCount = atomic.inc(FCount);
4   record(s, oldFCount);
5   ...
6   if ((--local_FCount)==0)
7     atomic_dec(FCount);
8 }

```

Figure 4: CCI-FunRe instrumentation. Line 5 represents the original function body, transformed so that all exits from the function pass through line 7.

multiple threads at the same time. The buggy implementation leads to logging failure when this function is re-entered. The $Reent_{ap_buffered_log_writer}$ predicate captures this mistake.

Of course, CCI-FunRe’s coarse granularity may limit accuracy. If function F performs multiple tasks, some parts of F may be safely reentrant while others are not. In this case, CCI will observe that $Reent_F$ is true whenever the program fails, but $Reent_F$ being true does not ensure failure. CCI can leverage such information to zoom into function F and get more accurate failure predictors. Section 5.2.2 discusses an example of this sort of diagnosis, currently performed manually, but potentially amenable to automation.

3.2.3 CCI-FunRe Instrumentation Strategy

In order to track re-entrance predicates, CCI maintains a global counter for each function F to indicate how many threads are executing F right now. This counter is updated at the entrance and the exit of F , as shown on lines 3 and 7 of Figure 4. A per-thread counter `local_FCount` is also maintained to prevent a thread from updating the global counter multiple times during recursive calls. Based on the global counter’s old value (`oldFCount`) at function entry, CCI records a single observation of the $Reent_F$ and $NonReent_F$ predicates on line 4. The counter update is implemented using atomic instructions.

3.3 CCI-Havoc Scheme

CCI-Havoc tracks whether the value of a given shared location changes between two consecutive accesses by one thread. Its name refers to the value-scrambling `havoc x` statement of Elmas et al. [11].

3.3.1 CCI-Havoc Instrumentation Sites

CCI-Havoc monitors each instruction I that might access a shared location g . Each such instruction can exhibit two possible behaviors at run time: either the value in g has not changed since this thread’s last access to g , or the value has changed. We say that I constitutes a single instrumentation site with two predicates: $Changed_I$ is true if the current value of g is different from its old value right after this thread’s last access to g , while $Unchanged_I$ is true if the prior and current values are the same. Each time I is executed, CCI records a

single true observation of either $Changed_I$ or $Unchanged_I$. In the example from Figure 1, if `len` is nonzero in thread 1, then $Changed_{cnt += len}$ is true in thread 2 because `cnt` has changed its value since the previous access by thread 2. In the example from Figure 2b, $Unchanged_S$ is true, since `mut` has the same value seen in the previous access by thread 2.

3.3.2 Diagnostic Potential of CCI-Havoc

Each CCI-Havoc predicate is only concerned with the state of the program as observed by a single thread. No cross-thread coordination is required. This makes CCI-Havoc instrumentation much simpler than that required for CCI-FunRe and CCI-Prev, and is similar to traditional CBI instrumentation.

Unlike traditional CBI predicates, CCI-Havoc predicates are closely related to atomicity violations, the most common root causes of crugs [27]. Previous studies [26, 32, 42] have shown that all atomicity violations that involve single variable can be categorized into four cases, shown in Figure 5. Three of these four cases are accurately captured by some CCI-Havoc predicate.

3.3.3 CCI-Havoc Instrumentation Strategy

CCI-Havoc instrumentation requires tracking the value held in each memory location right after each thread’s last access of that location. Given this information, CCI can evaluate CCI-Havoc predicates when needed as the instrumented program runs.

Figure 6 illustrates the instrumentation inserted by CCI at compile time. The instrumentation performs two tasks: (1) evaluating predicates on the present state, and (2) updating the history information for future use. Lines 1 and 2 of Figure 6 perform the first task. Function `test` uses a per-thread hash table to find the old value stored at `g` right after the last access to `g` from the current thread. The comparison between the old and the current values stored at `g` sets the local variable `changed` to true if and only if `g` has been changed. Subsequently, the value of `changed` is used to record CCI-Havoc predicates as shown on line 2. The `insert` call on line 4 updates the same per-thread hash table to map `&g` \mapsto `g`, i.e., to record the current value of `g` as the last value seen by this thread at address `&g`.

Note that, unlike CCI-Prev and CCI-FunRe, all information used by CCI-Havoc is thread-local. Therefore, evaluating CCI-Havoc predicates requires no locking or synchronization of any kind, which helps the performance and scalability of CCI-Havoc as shown in Section 5. Actually, there is a (benign) race in our instrumented code: theoretically, the value of `g` could be changed by other threads between lines 1 and 3. Fortunately, this type of low-probability noise is easily handled by CCI’s supporting statistical models [25] and therefore does not perceptibly affect CCI’s failure diagnosis.

The current implementation of CCI-Havoc uses the similar lightweight analysis as that in CCI-Prev to decide which are possibly-shared memory locations. This can be enhanced by escape analysis in the future.

<i>Thread 1</i> A: read g	<i>Thread 2</i> C: write g	<i>Thread 1</i> A: read g	<i>Thread 2</i> C: write g	<i>Thread 1</i> A: write g	<i>Thread 2</i> C: write g	<i>Thread 1</i> A: write g	<i>Thread 2</i> C: read g
B: read g		B: write g		B: read g		B: write g	
Case 1: read followed by read, interleaved by a write		Case 2: read followed by write, interleaved by a write		Case 3: write followed by read, interleaved by a write		Case 4: write followed by write, interleaved by a read	

Figure 5: Possible cases of single-variable atomicity violation. Cases (1)–(3) are captured by $Changed_B$ predicates.

```

1 changed = test(&g, g);
2 record(s, changed);
3 access(g);
4 insert(&g, g);

```

Figure 6: CCI-Havoc instrumentation. The original program code, represented here as `access(g)` on line 3, was either a read or a write of possibly-shared memory location `g`.

4. CCI Sampling

Sampling can be highly effective in achieving the low overheads that production usage demands. The challenge is to find suitable sampling strategies for each type of predicate. A careless or inappropriate approach to sampling cannot guarantee the correctness of predicate evaluation and may sample 90% of the execution with only 10% of the predicates collected.

Like CBI, CCI randomly decides which code regions to sample at run time. The sampling rate can be adjusted to control the imposed overhead. Unlike CBI, CCI sampling must make several extra decisions to maintain correctness and good coverage:

Thread-coordinated versus independent sampling. When using predicates (e.g., CBI, CCI-Havoc) that concern themselves with data and control activity within single threads, each thread can make local, independent decisions about when to start/stop the sampling. However, CCI predicates that monitor interactions among multiple threads require inter-thread coordination.

Length of each sampling period. In CBI, each sampling “period” only lasts for one statement. This works well for CBI predicates, as each observes the program state at one execution point. However, this is not suitable for CCI predicates that require considering multiple execution points together.

Correctness of predicate evaluation. Monitoring interleaving patterns may require history information (e.g., regarding the previous access to this variable). Unfortunately, sampling never presents a complete history of program execution. Correct evaluation of certain predicates demands a hybrid approach with both unconditional and sampled instrumentation.

The remainder of this section discusses the design and implementation of appropriate sampling strategies for the three types of CCI predicates presented earlier.

```

1 if (gsample) {
2   lock(glock);
3   changed = test_and_insert(&g, curTid, &stale);
4   record(stale ? s1 : s2, changed);
5   access(g);
6   gLength++;
7   unlock(glock);
8   lLength++;
9   if ((iset == curTid && lLength > lMAX)
10      || gLength > gMAX) {
11     clear ();
12     iset = unusedTid;
13     gsample = false;
14   }
15 } else {
16   access(g);
17   [[ gsample = true; iset = curTid; lLength=gLength=0;]]?
18 }

```

Figure 7: Sampled CCI-Prev instrumentation. “[[...]]?” marks a block of code that is randomly sampled using traditional CBI sampling methods.

4.1 CCI-Prev Sampling

CCI-Prev requires thread-coordinated, bursty sampling.

Recall that CCI-Prev detects intervening accesses by any other active thread. Thus, we must activate sampling at roughly the same time in all threads in order to collect accurate CCI-Prev predicates. If sampling were not thread-coordinated, then any non-sampling thread could “sneak in” and access shared data without notifying other sampling threads that it had done so.

To implement thread-coordinated sampling, CCI uses one shared global variable `gsample` to control whether to run instrumentation in all threads. Once `gsample` is set/unset in one thread, all threads begin/end their sampling. Figure 7 shows how the basic instrumentation from Figure 3 is augmented with sampling. CCI uses the basic random sampling framework of CBI to set `gsample` at line 17 to turn on sampling. Once sampling is turned on the instrumentation code at lines 2–14 is enabled in all threads. When sampling is turned off, lines 16–17 are executed.

The length of each sampling period is critical for CCI-Prev sampling. A sampling period must last long enough time to cover at least one pair of consecutive accesses to a shared

memory location in order to accurately record one CCI-Prev predicate. However, the sampling period cannot be too long, or else performance will suffer. Park et al. [32] have shown that temporal locality exists in crugs, especially atomicity violation bugs. One previous study has observed that the atomic regions involved in typical atomicity violation bugs range from 500 to 750 instructions [28]. Based on this, our current CCI prototype uses fixed thresholds to end sampling periods: 100 global accesses executed by the thread that starts the sampling or 10,000 global accesses executed by all threads, whichever occurs first. Of course, there could be other schemes to decide when to stop a sample period, such as variants of Hirzel and Chilimbi’s bursty tracing [19]. We plan to explore these alternatives in the future.

Lines 9–14 of Figure 7 demonstrate how to end a sampling period. Our implementation keeps the identity of the thread that turns on sampling by a global variable `iset`, which is set at line 17 along with `gsample`. That thread can turn off the sampling by clearing the `gsample` flag when the per-thread sampling length threshold is reached, as shown on line 9. Other threads can also turn off the sampling if the global sampling length threshold is reached first, as shown on line 10.

Finally, we need to revise predicate evaluation to maintain correctness in the presence of sampling. Specifically, we need to differentiate information collected during earlier sampling periods (referred to as *stale* information) and that during the current period (referred to as *fresh* information). When an instruction I conducts the first access to a memory location g during a sampling period, CCI does not have fresh information about preceding accesses to g and cannot guarantee the correctness of $remote_I$ and $local_I$.

Our implementation uses a generation counter to differentiate fresh from stale entries in CCI-Prev’s hash table. The generation count of each new entry is set to be the current generation counter at the time of insertion. At the end of every sampling period, function `clear` is called (line 11 in Figure 7) to increment the generation counter. Thus, a hash table entry is stale if its generation count is smaller than the current generation counter.

Only fresh entries are used to update *remote* and *local* predicates. To fully leverage the sampled information, each instrumented instruction maintains a secondary instrumentation site which considers stale hash table entries. Our rationale is that wrong predicates will be pruned out with high probability through CBI-style statistical analysis anyway. Therefore, keeping these secondary predicates can exploit more run-time information without increasing false positives. This design decision affects lines 3–4 in Figure 7. Function `test_and_insert` sets `stale` to true if the entry corresponding to `&g` is stale. Instrumentation site `s1` uses the stale entry and records the predicate, while site `s2` always uses information gathered in the current sampling period.

```

1 F (...) {
2   if ((local_FCount++)==0)
3     oldFCCount = atomic_inc(FCCount, curTid);
4   [[ record(s, oldFCCount); ]]?
5   ...
6   if ((--local_FCount)==0)
7     atomic_dec(FCCount, curTid);
8 }

```

Figure 8: Sampled CCI-FunRe instrumentation. “[[. . .]]?” marks a block of code that is randomly sampled using traditional CBI sampling methods.

4.2 CCI-Havoc Sampling

CCI-Havoc sampling requires thread-independent, bursty sampling.

CCI-Havoc predicates compare the values of one memory location at two nearby execution points in one thread. Since each predicate only cares about one thread, the sampling decision can be made independently by each thread, as in CBI. Since each predicate involves more than one execution point, each sampling period needs to reach certain length, as in CCI-Prev. In our current prototype, guided by previous study of typical atomic-regions’ length [28], each sampling period ends when the sampling thread conducts more than 100 accesses to shared memory locations.

The implementation of CCI-Havoc sampling uses traditional CBI mechanism to randomly turn on a thread-local variable that indicates sampling is active. Subsequently, a thread-local counter is incremented whenever this thread accesses a shared memory location. Sampling is turned off when the counter exceeds the predefined 100-access threshold. The thread-local hash table that hold mappings from memory locations to their old values is periodically flushed in a similar way to that done for the global hash table in CCI-Prev.

4.3 CCI-FunRe Sampling

CCI-FunRe requires a mixture of unconditional and CBI-style (thread-independent, non-bursty) sampled instrumentation.

The special challenge of CCI-FunRe is that we must instrument *all* invocations of F in order to guarantee the correctness of $Reent_F$. This challenge is different from CCI-Prev and CCI-Havoc, where correctness is guaranteed as long as fresh information is used. For CCI-FunRe, when fresh information indicates that $Reent_F$ is false, this may not reflect reality, because a non-sampled invocation to F may have occurred before the current sampling period began.

This problem has two potential solutions. The first is to statically divide functions to groups and monitor all invocations of functions from one group at each user’s site. The disadvantage of this approach is the burden of deploying differently-instrumented executables for different users. The second solution is to perform unconditional (always-active) instrumentation for all function invocations (lines 3 and 7 in

Program	KLOC	Symptoms	Runs	
			Total	Failed
Apache-1	333	corrupted log	3,000	1,372
Apache-2	333	crash	3,000	1,566
Cherokee	83	corrupted log	3,000	1,705
FFT	1.3	wrong output	3,000	1,766
LU	1.2	wrong output	3,000	1,179
Mozilla-JS-1	107	crash	3,000	1,660
Mozilla-JS-2	107	wrong output	3,000	1,493
Mozilla-JS-3	107	crash	3,000	1,507
PBZIP2	2.1	crash	3,000	1,350

Table 1: General characteristics of buggy test subjects. “KLOC” is total program size in thousands of lines of code.

Figure 8) and only apply sampling to predicate evaluation and recording (line 4). The advantage is that only one instrumented executable is needed and the overhead of production-run monitoring can be adjusted via the sampling rate as effectively as before. The disadvantage is that unconditional overhead due to lines 3 and 7 cannot be controlled through the sampling rate. We use the second solution in our current prototype, and leave the first for future work.

5. Experimental Evaluation

5.1 Methodology

We have implemented CCI and evaluated it to answer two key questions: (1) how accurate is CCI in reporting root causes of concurrent program failures, and (2) what is the performance overhead of CCI monitoring? To get a better understanding of CCI’s failure diagnosis capability, we tried two conventional CBI instrumentation schemes for comparison: one that records the directions of conditional branches, and one that monitors the relative values of same-typed pairs of scalar variables. To assess the effectiveness of CCI sampling, we compared the performance of CCI with and without sampling. To demonstrate the difference between race detection and CCI failure diagnosis, we also did experiments with Helgrind [2], a state-of-practice race detector.

Our experiments were carried out on quad-core Intel machines using several widely used C applications with real bugs. In all experiments, we added randomly-executed thread yield calls in the source code in order to make the program fail more frequently. These random yields will not affect the quality of CCI evaluation; they merely change the ratio of successful to failing runs. In practice, failures would be much less common in the field. However, as enough failures have occurred, one could analyze these failures along with a randomly-selected subset of successes to achieve a similar mix.

Table 1 shows some characteristics of the buggy test subjects and the experimental runs. The “Mozilla-JS” test subject is a standalone JavaScript engine from the Mozilla web browser.

Program	CBI	CCI-Prev	CCI-Havoc	CCI-FunRe
Apache-1	-	✓ 1	✓ 1	✓ 1
Apache-2	-	✓ 1	✓ 1	-
Cherokee	-	-	✓ 2	-
FFT	-	✓ 1	-	-
LU	-	✓ 1	-	-
Mozilla-JS-1	-	-	✓ 2	✓ 1
Mozilla-JS-2	-	✓ 1	✓ 1	✓ 1
Mozilla-JS-3	-	✓ 2	✓ 1	✓ 1
PBZIP2	-	✓ 1	✓ 1	-

Table 2: Overall failure diagnosis results. “✓ n ” indicates that the n^{th} highest ranked predictor captures the root cause, while “-” indicates that neither of the **top two** predictors is useful.

We target an effective sampling rate close to $1/100$ as recommended by Liblit et al. [24]. Specifically, for CCI-Prev and CCI-Havoc that use 100-access bursty sampling periods, we use start sampling periods with probability $1/10,000$. For CCI-FunRe, which uses single-sample “bursts”, the sampling rate is exactly $1/100$. We use the iterative ranking and elimination model of Liblit et al. [25] to mine collected data for failure predictors. Failure predictors discovered by the statistical model must be correlated with a positive increase in failure likelihood with at least 95% confidence.

Tables 3–5 visualize analysis results using *bug thermometers*, one per predictor selected by the statistical model [25]. The width of a thermometer is logarithmic in the number of runs in which the predicate was observed. The black band on the left denotes the *context* of the predicate: the probability of failure given that the predicate was observed at all, regardless of whether it was true or false. The dark gray or red band denotes the 95%-certain increase in the probability of failure given that the predicate was true. The light gray or pink band shows the additional increase that is estimated but not at least 95% confident. A large dark gray/red area indicates that the predicate being true is highly predictive of failure, and a small light gray/pink band indicates that this prediction carries high confidence. Any white space at the right edge of the band indicates the number of successful runs in which the predicate was observed to be true: a measure of the bug predictor’s non-determinism.

Complete analysis results allow some interactivity, and include more details such as the precise source file name and line number on which the predictor was observed. These features are helpful in a programmer’s hands, but we omit them here to simplify presentation.

5.2 Failure Diagnosis Results

5.2.1 Overall Results

Table 2 shows the overall failure diagnosis results of three CCI schemes together with a baseline CBI scheme, all under the roughly $1/100$ effective sampling rate described in Section 5.1. We consider diagnosis successful if either of the

topmost two predictors clearly describes the conditions for failure and would lead a developer directly to the bug.

As we can see, CCI-Prev, CCI-Havoc, and CCI-FunRe can all help diagnose real-world crugs. Their top-ranking predictors can help explain 7, 7, and 4 out of the tested 9 crug failures, respectively. By contrast, the baseline CBI cannot diagnose any crug failures. In some cases (Apache-1, Mozilla-JS-1, and LU), all conventional CBI predicates are eliminated due to low ($< 95\%$) confidence that they behave differently in failing versus successful runs. In other cases, some CBI predicates have statistical correlation with failure, but none of them, no matter how high- or low-ranked, are relevant to the failures. This affirms our earlier claim that conventional CBI is ill-equipped to diagnose crugs.

Among the three CCI schemes, the failure diagnosis capabilities of CCI-Prev and CCI-Havoc are especially good. CCI-Prev can diagnose not only atomicity violation bugs (Apache-1, Apache-2, Mozilla-JS-2, and Mozilla-JS-3) but also order violation bugs and races (FFT, LU, and PBZIP2). CCI-Havoc cannot detect order-violation bugs, but it successfully diagnoses all atomicity violation bugs in our experiments with high confidence. CCI-FunRe shows the weakest diagnosis capability due to its coarse granularity.

In order to understand whether sampling has any impact on above diagnosis results, we repeat the experiments without sampling. We find exactly the same results except for two cases: the Mozilla-JS-1 failure can be successfully diagnosed by CCI-Prev without sampling; the Mozilla-JS-3 failure can be successfully diagnosed by CBI without sampling. For CCI-Prev, the reason is that the Mozilla-JS-1 failure involves a shared memory access and a remote preceding access that are so far away from each other that they do not fit into one bursty sampling period. This shows that the fixed size bursty windows could lead to false negatives in some cases. As mentioned in Section 4.1, future work can randomize the bursty-sampling window size within some suitable distribution. In all other cases, sampling never hurts CCI's diagnoses, thanks to the statistical models previously developed by Liblit et al. [25]. For the CBI case, the reason is that some program sites and the corresponding predicates may not reach statistical significance under sparse sampling. Thus, all but two '-' marks in Table 2 are caused not by the information loss due to sampling, but rather by the inherent limitations of each instrumentation scheme. We explore this further below.

5.2.2 Case Studies

We now use case studies to explain the differing diagnosis capabilities of CCI-Prev, CCI-Havoc, and CCI-FunRe. We examine just a few of the bugs from our experiments in detail here; Sections 5.2.1 and 5.3 respectively report diagnosis and performance results across the entire suite of buggy programs.

Apache HTTP Server Case 1 is a case where all three CCI schemes successfully diagnose the failure.

In this experiment we use CCI to diagnose a non-deterministic log corruption problem in Apache. This bug (illustrated in Figure 1) was originally reported by Apache users Sussman and Trawick [40] on the Apache Bugzilla bug tracker. Our experiment consists of 3,000 runs based on this bug report. Each run starts the Apache HTTP Server, downloads two files in parallel ten times, then stops the server. Each run is labeled as a failure if the server's log file is corrupted and a success otherwise.

Table 3 lists the top-ranked bug predictor from each CCI scheme. As we can see, all CCI schemes successfully identify the root cause of this Apache failure. CCI-Prev points out that the failure is related to an unexpected remote access to `cnt` preceding the read of `cnt` at `cnt += len` in function `ap_buffered_log_writer`. CCI-Havoc points out that the failure is related to an unexpected change to `cnt` between its two uses in this same function; CCI-FunRe identifies reentrant execution of this same function as the reason for the failure.

This Apache failure and the three Mozilla failures are the four failures in our experiments that CCI-FunRe can successfully diagnose. In all cases, the involved functions are relatively short. Each performs a simple task (e.g., write to the global log) that should not be carried out without synchronization.

Cherokee is a case where only CCI-Havoc successfully diagnoses the failure. Table 4 shows the top two predictors of CCI-Havoc. Actually, the first predictor points to **a crug that we were previously unaware of**; the second predictor successfully explains the random log corruption problem that we tried to diagnose.

Figure 9 illustrates this bug, originally reported by Cherokee users. Its symptom is non-deterministic log corruption, similar to that of the Apache bug discussed above. However, only CCI-Havoc successfully diagnoses Cherokee's variant of this problem. As Figure 9 shows, the Cherokee log (`g_buf`) is corrupted whenever the buffer index `g_buf->len` is modified between `S1` and `S2`. CCI-Havoc correctly identifies the failure predictor as shown in Table 4.

CCI-Prev was unable to identify `remoteS2` as a failure predictor because `S2` could be preceded by a read to `g_buf->len` from a different thread, which will not cause the failure. Ignoring the access type of preceding accesses causes this false negative. Future work can extend CCI-Prev to collect four types of predicates `remote → write`, `remote → read`, `local → write`, and `local → read`, in order to diagnose failures similar to this Cherokee one.

CCI-FunRe did not identify function `update_guts` as a failure predictor, simply because `update_guts` is a large, complex function. The majority of its actions do not use `g_buf` and can be safely executed by multiple threads simultaneously. CCI-FunRe could diagnose this failure if its monitoring granularity were smaller than one function. We tried splitting `update_guts` to several monitoring blocks using preexisting

Scheme	Thermometer	Predicate	Function
CCI-Prev		cnt remote \rightarrow read	ap_buffered_log_writer
CCI-Havoc		cnt changed \rightarrow read	ap_buffered_log_writer
CCI-FunRe		reentrant call	ap_buffered_log_writer

Table 3: Top predictor from each scheme for Apache HTTP

Thermometer	Predicate	Function
	buf2- \rightarrow len changed \rightarrow read	buffer_add_buf
	g_buf- \rightarrow len changed \rightarrow read	update_guts

Table 4: Top CCI-Havoc predictors for Cherokee

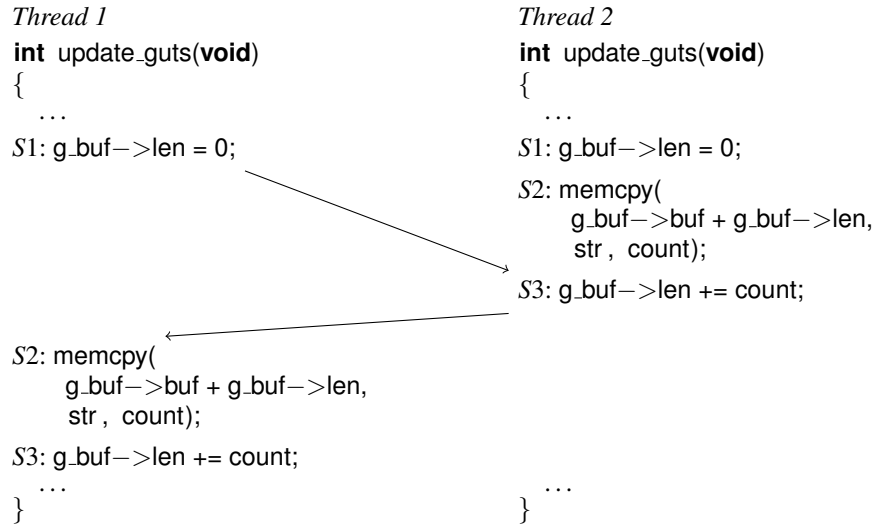


Figure 9: Simplified illustration of the Cherokee bug

return statements as block-boundaries This lets CCI-FunRe uncover an accurate failure predictor. Our splitting was performed manually, but could certainly be automated as part of a process of iterative instrumentation refinement.

FFT and LU from SPLASH-2 are two cases where only CCI-Prev successfully diagnoses the failures.

Failures in these two applications are non-deterministic wrong outputs of timing statistics. They are caused by order-violation bugs, introduced by wrong implementation of the `WAIT_FOR_END` macro in the `c.m4.ia32` file used. The missing macro allows a race between two sets of operations: (1) assignments at the end of the worker thread to global variables that maintain the timing information, and (2) the printing of these global variables at the end of the parent thread. Our experiment consists of running the given program 3,000 times and marking each run as a failure if the printed execution finish-time is invalid (i.e., negative or zero).

CCI-FunRe and CCI-Havoc cannot effectively diagnose these two failures because the root causes have nothing to do with function reentrance or unexpected value changes be-

tween two uses. However, CCI-Prev does correctly diagnose these failures benefiting from the secondary instrumentation sites that leverage stale information per Section 4.1. The top CCI-Prev predictor, shown in Table 5, corresponds to the assignment at the end of the worker thread which stores the correct timing information into `initdonetime`. Only during failure runs, this assignment is preceded, instead of followed, by a read access to `initdonetime` from the parent thread. The second-ranked predictor of FFT can also explain the failure. It shows that failure occurs when the read access to `initdonetime` in the parent thread is not preceded by a remote assignment to `initdonetime`.

5.3 Performance Results

Table 6 shows the overheads of each instrumentation scheme with and without sampling for the evaluated applications (without inserting any random yields). The two bugs in Apache require different source code configurations and have different performance features. The three Mozilla bugs can be triggered with the same input and source code. Therefore, we only present one number in the table.



Thermometer	Predicate	Function
	Global \rightarrow initdonetime remote \rightarrow write	SlaveStart
	Global \rightarrow initdonetime local \rightarrow read	main

Table 5: Top CCI-Prev predictors for FFT

	No Sampling			Sampling		
	Prev	FunRe	Havoc	Prev	FunRe	Havoc
Apache-1	62.6%	1.1%	27.4%	1.9%	1.8%	2.8%
Apache-2	8.4%	0.2%	4.2%	0.5%	0.2%	0.4%
Cherokee	19.1%	0.3%	2.1%	0.3%	0.4%	0.0%
FFT	169 %	72.8%	33.5%	24.0%	30.0%	5.5%
LU	57827 %	1682 %	1693 %	949 %	926 %	8.9%
Mozilla-JS	11311 %	123 %	7587 %	606 %	97.0%	356 %
PBZIP2	0.2%	0.3%	0.2%	0.2%	0.2%	0.2%

Table 6: Run-time overheads. Overheads below 10% are **boldfaced**.

Overall, sampling significantly decreases CCI’s monitoring overhead. CCI offers the potential of low-overhead crug diagnosis in production runs for many of the applications in our experiments.

Specifically, if we use 10% run-time overhead as a threshold for deployment, then without sampling overheads are already low enough to deploy CCI-Prev in two out of seven applications, CCI-Havoc in three applications, CCI-FunRe in four applications. CCI-FunRe is so simple a scheme that it has negligible overhead (around 1%) for all I/O intensive applications (Apache-1, Apache-2, Cherokee, and PBZIP2) in our experiments. Unfortunately, memory-access intensive applications, such as the Mozilla JavaScript Engine and SPLASH2 applications, still incur huge monitoring overhead for all three schemes.

Sampling significantly shrinks overheads for CCI-Prev and CCI-Havoc. With sampling, CCI-Prev achieves small enough overhead to deploy in two more applications (Apache-1 and Cherokee). CCI-Havoc is even better: overhead drops below 10% for three more applications, pushing the number of deployable applications to six out of seven. Sampling allows all CCI schemes to achieve negligible ($< 3\%$) overhead for all I/O intensive applications in our experiments. It also helps CCI-Havoc achieve small ($< 10\%$) overhead for all evaluated applications except for the Mozilla JavaScript Engine. The large overhead in the later is caused by its extremely intensive heap accesses and loops. CCI pessimistically assumes that any pointer-crossing operation might touch shared memory, but this is clearly an over-approximation. We expect that static thread-escape analysis and CCI-aware loop unrolling will significantly decrease overhead in the future.

Comparing the different CCI instrumentation schemes, CCI-Prev has the worst performance because it uses locks and global flags (i.e., `gsample`) to coordinate sampling and predicate evaluation across all threads. CCI-FunRe has the

best performance without sampling, but benefits the least from sampling, per Section 4.3. CCI-Havoc has a simpler design than CCI-Prev and is also easier to sample than CCI-FunRe. Thus, CCI-Havoc achieves the best balance between performance and failure diagnosis among all three schemes.

What is the minimum overhead CCI can deliver? Theoretically, lower sampling rates yield lower overheads. However, the overhead can never reach zero. In both CCI-Prev and CCI-Havoc, the run-time environment maintains a random number generator, checks the “`gsample`” flag if necessary, and conducts a countdown in order to support random sampling, which determines the lower-bound of CCI monitoring overhead. CCI-Prev’s minimum overhead for FFT, LU, and Mozilla-JS is 6.6%, 2.8%, and 562%, respectively; CCI-Havoc’s minimum overhead for these three applications are 4.4%, 0.8%, and 298%, respectively. For CCI-Havoc’s thread-independent sampling, we can skip the “`gsample`” flag check when we know that sampling is currently off and will not be turned on during several forthcoming sites. Thus we see lower overheads for this scheme. In general, this minimum overhead increases with the density of instrumentation sites and loops in the program. Mozilla-JS has high density of both and therefore has high minimum overhead. As discussed above, we expect that improved static analysis will help applications like Mozilla-JS in the future.

What type of sampling can help CCI-FunRe? As discussed in Section 4.3, one sampling strategy that can further decrease the overhead of CCI-FunRe is to statically split functions into multiple groups and release multiple versions of software with each version monitoring one group. We have implemented this scheme, and find that it does help CCI-FunRe to significantly decrease its monitoring overhead on Mozilla-JS to less than 15%. However, it still has drawbacks. For example, a single small yet frequently-invoked function

Program	# of False Positives	Find Failure- Relevant Race?	Overhead
Apache-1	44	✓	57%
Apache-2	81	✓	29%
Cherokee	10	✓	212%
FFT	27	-	20,233%
LU	22	-	41,172%
Mozilla-JS-1	9	✓	45,957%
Mozilla-JS-2	24	✓	45,957%
Mozilla-JS-3	19	✓	45,957%
PBZIP2	11	✓	4,803%

Table 7: Bug detection results of Helgrind race detector

could become a performance bottleneck with CCI-FunRe, a possibility not addressed by this alternative sampling scheme.

How scalable are CCI sampling schemes? Our experiments show that CCI-Havoc has excellent scalability. We find that with a given workload, adding more threads *reduces* CCI-Havoc overhead in our SPLASH2 experiments. This is because evaluation of CCI-Havoc predicates is completely independent by thread and therefore can be parallelized perfectly. The scalability of CCI-FunRe and CCI-Prev is not as good as CCI-Havoc, making CCI-Havoc the instrumentation scheme of choice for very-highly-concurrent applications.

5.4 Comparison With Helgrind Race Detector

To demonstrate the difference between race detectors and CCI, we applied Helgrind, a state-of-practice happens-before race detector from Valgrind [2], to all 9 bugs with exactly the same inputs and experiment settings as those used by CCI. The results are shown in Table 7.

Helgrind reports many false positives, ranging from 9 in Mozilla-JS-1 to 81 in Apache-2, which could cost significant developer effort during failure diagnosis. These false positives can be categorized into two types. The first type are true races, but are considered harmless by programmers and are intentionally left there. For example, all 9 false positives in Mozilla-JS-1 belong to this type. Prior work reports similar results [7, 31], and in general no race detector can avoid these false positives as they genuinely are true races, just not failure-inducing ones. The second type of false positives are not truly races at all. Helgrind mistakenly reports them because it only recognizes synchronizations implemented by the pthread library. For example, 16 out of the 19 false positives in Mozilla-JS-3 are actually well-synchronized by ad-hoc condition variables implemented by the programmers. A more accurate race detector could avoid reporting these.

Helgrind is able to report races that are related to the software failure for 7 out of the 9 bugs in our study. However, due to the different design purposes, the information provided by Helgrind is usually less useful to failure diagnosis than that provided by CCI. For example, for the Cherokee bug shown in Figure 9, Helgrind reports a data race between $S1$ and $S3$. Following this clue, programmers might put $S1$ into a

critical region and $S3$ into another critical region protected by the same lock. This would eliminate the data race, but would not fix the problem. By contrast, CCI-Havoc reports that Cherokee tends to fail when the global variable $g_buf \rightarrow len$ is modified between $S1$ and $S2$. This is exactly the root cause of the failure and can help developers to fix the bug.

Finally, Helgrind has a huge overhead that is obviously not suitable for production run. This is part of the motivation of CCI and other sampling-based race detectors [5, 29].

6. Related Work

Pre-deployment tools for detecting races and atomicity violations fall into two categories: static and dynamic. Static approaches [12, 17, 21] are conservative and must consider all potential races. A problem with using static analysis is that it is difficult to distinguish benign races from those that can genuinely cause failures. Benign races occur, for example, in test-and-set-lock operations and performance counter updates. Scalability of analyses to target large programs is also problematic.

Many dynamic analysis tools have been proposed to detect data races and atomicity violation bugs [13, 14, 26, 36]. All have high run-time overheads (around $25\times$ for C applications) which make them impractical for post-deployment use. Furthermore, each of these tools targets only a specific class of crugs viz. either atomicity violations [13, 26] or data races [36], and often assume a particular synchronization mechanism. For example, the lockset analysis used in Eraser [36] applies only to lock-based multi-threaded programs. CCI targets the root causes of a wide variety of software failures caused by not just data races but also atomicity violations and other types of crugs and is agnostic to which particular synchronization mechanism is used. Furthermore, by focusing on predictors for genuine failures, CCI avoids the false positives caused by benign races which plague other static and dynamic approaches.

Some of these issues also apply to recent approaches that use sampling to improve the performance of race detection, such as LiteRace by Marino et al. [29] and PACER by Bond et al. [5]. CCI shares the same sampling philosophy with LiteRace and PACER. However, they are designed for different purposes: CCI targets failure diagnosis while LiteRace and PACER focus on race detection. Because of this difference, CCI covers a wider variety of synchronization problems than LiteRace and PACER, and is explicitly agnostic with respect to synchronization mechanisms. CCI also leverages its statistical model and failure information to achieve high accuracy, while LiteRace and PACER would cause many false positives due to benign races if used in failure diagnosis, similar to Helgrind race detector discussed in Section 5.4. In addition, the sampling techniques used are also different. LiteRace uses adaptive, thread-independent, bursty sampling; PACER uses thread-coordinated, bursty sampling. Our three CCI schemes cover different sampling design choices, in-

cluding both thread-independent, thread-coordinated, and different bursty designs.

Our earlier and shorter version of this work [41] designed one predicate (here identified as CCI-Prev) and one basic thread-coordinated sampling strategy. The present work enhances the previous CCI-Prev sampling scheme with tunable, bursty sample lengths. More importantly, the present work designs and thoroughly evaluates a wider range of predicates and sampling strategies; the prior work by Thakur et al. can now be seen as one point in the much richer design space explored herein.

Previous work also improves the performance of race detection by replacing heavyweight vector clocks with an adaptive lightweight representation in FastTrack [14] and using adaptive coarse granularity memory monitoring in RaceTrack [46]. These performance enhancing techniques are orthogonal to the sampling techniques used in CCI. It is conceivable to further improve their performance with sampling. CCI can also use adaptive monitoring granularity to further improve its performance. Of course, since CCI is looking at failure diagnosis instead of race detection, the design tradeoffs will be different than those in RaceTrack.

Much prior research has focused on testing concurrent programs, such as testing based on synchronization coverage [6], context-bounded testing [30], race-directed random testing [38], and unit testing[33]. Record-and-replay for multi-core machines could also aid in debugging concurrent programs. Unfortunately, existing proposals are not practical for production usage due to high overhead (around 10× slowdown [10, 22]) or reliance on non-existing hardware [20]. Recent approaches aim to not detect, but to automatically avoid or tolerate certain kinds of crugs [4, 28, 34, 35, 43, 45]. This is orthogonal to our goal of diagnosing the root causes of concurrent software failures.

DefUse[39] proposes a family of define-use related invariants to capture dynamic data-flow and detect software bugs. As DefUse is used for in-house testing while CCI is used in the field, they have focused on different challenges and proposed different techniques. CCI predicates are different from DefUse invariants. To maintain simplicity, CCI-Prev and CCI-Havoc are *intentionally* ignorant of the access type (read or write) and hence also ignorant of define-use data dependencies. CCI also uses sampling techniques to lower overheads, and a statistical model that handles the resulting noise/inaccuracy.

CCI is based on the Cooperative Bug Isolation (CBI) project [23, 25], which uses a sampling-based monitoring framework to ensure that run-time overheads are low, and uses statistical techniques on the collected data to infer likely root causes from this sparsely-sampled data. Subsequent work has further refined the CBI paradigm to find root causes of more complex bugs [3, 8], but our experiments have demonstrated that crugs demand a new approach.

7. Conclusion

We have described CCI, a low-overhead, scalable strategy for root-cause analysis of crugs. We have implemented the system and shown our technique to be effective at finding bugs in several large, real-world applications. Our approach intentionally tracks far less information than exhaustive dynamic detectors. Combined with novel approaches to cross-thread sampling, this allows CCI to achieve very low overheads, making it practical for use in production environments. At the same time, the data collected is sufficient to isolate root causes of failures that are invisible to prior statistical debugging work. In the future we plan to extend this work by exploring other instrumentation schemes that track different concurrency events, by experimenting with other thread-aware sampling mechanisms, and by augmenting CCI's dynamic approach with complementary static analyses.

References

- [1] The Apache Software Foundation. Apache HTTP Server Project. <http://httpd.apache.org/>, 2009.
- [2] C. Armour-Brown, J. Fitzhardinge, T. Hughes, N. Nethercote, P. Mackerras, D. Mueller, J. Seward, B. V. Assche, R. Walsh, and J. Weidendorfer. *Valgrind User Manual*. Valgrind project, 3.5.0 edition, Aug. 2009. <http://valgrind.org/docs/manual/manual.html>.
- [3] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *ISSTA*, 2007.
- [4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA*, 2009.
- [5] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.
- [6] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPOPP*, 2005.
- [7] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.
- [8] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE*, May 2009.
- [9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.
- [10] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.
- [11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.
- [12] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5), 2003.
- [13] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
- [14] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

- [15] J. Gilchrist. PBZIP2: Parallel BZIP2 Data Compression Software. <http://compression.ca/pbzip2/>, 2009.
- [16] P. Godefroid and N. Nagappan. Concurrency at Microsoft – an exploratory survey. In *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.
- [17] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
- [18] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.
- [19] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.
- [20] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multi-processor replay. *CACM*, 2009.
- [21] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, 2007.
- [22] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4), 1987.
- [23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
- [24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of Cooperative Bug Isolation. In *RAMSS*, 2004.
- [25] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
- [26] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. In *ASPLOS*, 2006.
- [27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, March 2008.
- [28] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1), 2009.
- [29] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.
- [30] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
- [31] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.
- [32] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.
- [33] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.
- [34] S. Rajamani, G. Ramalingam, V.-P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in concurrent programs. In *ASPLOS*, 2009.
- [35] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP*, 2009.
- [36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15, 1997.
- [37] SecurityFocus. Software bug contributed to blackout. <http://www.securityfocus.com/news/8016>, Feb. 2004.
- [38] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
- [39] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I Use the Wrong Definition? DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *OOPSLA*, 2010.
- [40] A. Sussman and J. Trawick. Corrupt log lines at high volumes. https://issues.apache.org/bugzilla/show_bug.cgi?id=25520, 2003.
- [41] A. Thakur, R. Sen, B. Liblit, and S. Lu. Cooperative Crug Isolation. In *WODA*, 2009.
- [42] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.
- [43] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.
- [44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.
- [45] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.
- [46] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.