

Cooperative Crug Isolation *

Aditya Thakur
adi@cs.wisc.edu

Ben Liblit
liblit@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison

Rathijit Sen
rathijit@cs.wisc.edu

Shan Lu
shanlu@cs.wisc.edu

ABSTRACT

With the widespread deployment of multi-core hardware, writing concurrent programs has become inescapable. This has made fixing concurrency bugs (or *crugs*) critical in modern software systems. Static analysis techniques to find crugs such as data races and atomicity violations are not scalable, while dynamic approaches incur high run-time overheads. Crugs pose a greater challenge since they manifest only under specific execution interleavings that may not arise during in-house testing. Thus there is a pressing need for a low-overhead program monitoring technique that can be used post-deployment.

We present Cooperative Crug Isolation (CCI), a low-overhead instrumentation technique to isolate the root causes of crugs. CCI inserts instrumentation that records occurrences of specific thread interleavings at run-time by tracking whether successive accesses to a memory location were by the same thread or by distinct threads. The overhead of this instrumentation is kept low by using a novel cross-thread random sampling strategy. We have implemented CCI on top of the Cooperative Bug Isolation framework. CCI correctly diagnoses bugs in several nontrivial concurrent applications while incurring only 2–7% run-time overhead.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*statistical methods*; D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids*

General Terms

Experimentation, Reliability

*Supported in part by AFOSR grants FA9550-07-1-0210 and FA9550-09-1-0279; LLNL contract B580360; NSF grants CCF-0621487, CCF-0701957, 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.

WODA '09, July 20, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-656-4/09/07 ...\$5.00.

Keywords

concurrency, statistical debugging, random sampling, bug isolation

1. INTRODUCTION

Concurrency bugs (such as data races [5, 23] and atomicity violation bugs [6, 17]) are among the most troublesome software bugs. Unlike sequential bugs, concurrency bugs manifest under only specific thread interleavings. This non-determinism makes concurrency bugs difficult to expose during in-house testing. As a result, many concurrency bugs slip into production runs and manifest at user sites. Making things even worse, many concurrency bugs can cause severe software failures, varying from data corruption to program crashes [8]. Concurrency bugs have caused real-world disasters in the past, such as the Northeastern Blackout of 2003. Software reliability is increasingly threatened by concurrency bugs, because of the prevalent use of multi-core hardware and the increasing use of concurrent programs. Thus, tools that can diagnose production-run failures in concurrent software are sorely needed.

To date, it has been extremely difficult to diagnose production-run software failures caused by concurrency bugs. There are no concurrency bug detection tools suitable for production usage, because existing tools either have huge overhead ($10\times$ – $100\times$ slowdown [23]) or require specialized hardware that does not yet actually exist [17]. Furthermore, it can be very hard for developers to reproduce field-detected software failures, because the manifestation of concurrency bugs demands special interleavings. Even if the bug-triggering input is known, it may still take developers several days to reproduce a concurrency bug [20]. As a result, failure diagnosis is a nightmare for the developers of concurrent programs.

Statistical debugging pioneered by the Cooperative Bug Isolation project (*CBI*) [3, 14, 16] aims to automatically pinpoint likely causes of failure. It achieves this goal through three steps. First, it instruments the buggy 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. Then, it collects information about program execution by collecting these predicate samples from both successful and failing runs. Finally, by applying statistical techniques on this aggregated information, CBI finds good bug predictors by identifying predicates that are highly correlated with failure. The CBI framework achieves low monitoring overhead by way of sparse random sampling of the instrumentation and by collecting information from many user sites.

Prior CBI work has not targeted concurrent programs and is not effective in diagnosing software failures caused by concurrency bugs. Failures caused by concurrency bugs are triggered by specific interleavings (i.e., execution order among concurrent memory accesses).

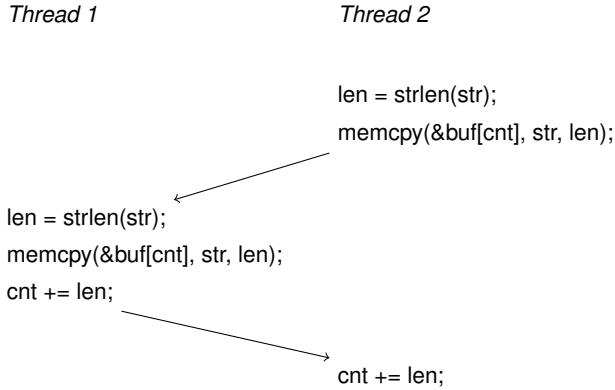


Figure 1: Atomicity violation bug from the Apache HTTP Server. The variables `buf` and `cnt` are both shared.

Many of these failures cannot be explained by the types of predicates collected by prior CBI work. Figure 1 shows a concurrency-bug example, simplified from a real-world bug in the Apache HTTP Server [1]. In this example, shared variable `cnt` is an index to the tail of a shared buffer `buf`. Every thread writes log message into the buffer based on the index. Unfortunately, programmers did not synchronize conflicting accesses to these two variables. As a result, under the interleaving shown in Figure 1, buffer update and index accesses from different threads compete with each other and lead to garbage data in the log.

Previous CBI tools fail to diagnose this Apache 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). The reason is that the software’s wrong behavior (garbage log data) can happen with normal variable values (e.g., the garbage log can happen when `cnt` remains in bound) and normal execution paths. This example illustrates the need to design new predicates in order to diagnose software failures due to concurrency bugs.

In this paper, we present Cooperative Crug Isolation (CCI), a low-overhead dynamic strategy for diagnosing production-run failures in concurrent programs. CCI builds on the CBI framework and employs the same underlying approach to pinpointing root causes of failure. We focus on the following two problems in order to extend CBI to address concurrency bugs:

What type(s) of predicates are suitable for diagnosing crug problems? A good predicate should balance the predictive power and performance. Poorly designed predicates may never be able to explain a software failure. Yet some run-time information, such as global execution orders, is quite costly to collect. Costly predicates must use low sampling rates in order to provide performance guarantees during production runs.

How can we track predicates that involve multiple threads?

In previous CBI work, predicate sampling decisions were always made independently on multiple threads. This is not suitable for interleaving-related predicates. Sampling must be coordinated across threads in order to track interleaving-related information.

To address the first problem, a new type of predicates is designed in CCI to record whether two successive accesses to a shared memory location were by the same thread or distinct threads. In particular, CCI instrumentation keeps track of which thread last accessed each shared memory location. At a particular program point I that accesses memory location l , the value of CCI predicate $remote_I$ is true if the thread that last accessed l differs from the current thread, and

is false otherwise. The other CCI predicate $local_I$ represents the opposite of $remote_I$.

Using this simple instrumentation scheme CCI is able to target a wide variety of concurrency bugs such as races and atomicity violations, as the occurrence of races or atomicity violations can usually affect the values of corresponding CCI predicates. Furthermore, CCI leverages statistical analysis to maintain much fewer false positives than many previous crug detection tools [23]. Many previous detection tools have high false positive rates, because races and atomicity violations could be benign. However, CCI predicates could be true when the program is well-synchronized. This does *not* cause false positives (i.e., reporting predicates as bug predictors even though they have nothing to do with the cause of the failure) in CCI, because the statistical analysis in CCI leverages information about whether a particular run succeeded or failed. CCI tries to correlate the values of the predicates with failures and figures out whether a predicate being true during execution can imply a failed execution. A false positive occurs only if a predicate was observed to be true in all (or most) failure runs and in no (or few) successful runs *and* had nothing to do with the actual cause of the failure. This is unlikely to occur in practice and we observe that the statistical analysis used in CCI prunes away most false positives and reports predictors which are precisely those which are correlated to the cause of the failure.

Similar to CBI, run-time overhead is kept low by sampling the instrumentation code. However, CCI requires sampling to be on in multiple threads simultaneously. Thus, CCI extends the current CBI sampling using a novel cross-thread random sampling strategy to ensure correct recording of its interleaving-related predicates.

Specifically, this paper makes the following contributions:

- We extend the CBI framework to track predicates that indicate whether two successive accesses to a memory location were by the same thread or not. This instrumentation scheme can help diagnose several types of crugs such as data races and atomicity violations.
- We extend the CBI sampling scheme to coordinate the predicate sampling from different threads in concurrent programs. This extension allows CCI to collect accurate interleaving information.
- We validate CCI by using it to identify the root causes of real-world failures in several concurrent applications, including Apache [1], PBZIP2 [7], and SPLASH-2 [28] benchmarks.

Preliminary experimental results show that CCI nicely complements CBI to diagnose concurrent program failures. The predictors reported by CCI can accurately point to the exact reason for the problem. On the other hand, previous CBI tools are unable to provide any predictors for these two applications. Furthermore, CCI achieves this excellent failure diagnosis with small run-time overhead (mostly within 10%), thanks to its sampling mechanism.

The remainder of this paper is organized as follows. Section 2 provides an overview of the CBI framework. Section 3 describes our instrumentation scheme in detail. We describe the experimental results in Section 4. We describe previous approaches in Section 5. Section 6 concludes and suggests future directions for this work.

2. BACKGROUND

CCI builds upon the Cooperative Bug Isolation (CBI) [14, 16] approach and framework. CBI is a low-overhead statistical debugging technique. It collects information about program execution from

both successful and failing runs and applies statistical techniques to identify the likely cause of the software failure.

CBI carries out a source-to-source transformation of the program which adds instrumentation code to the original program to collect 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:

1. Branches: Each branch is an instrumentation site. Two predicates indicating whether the true or false branches were taken are associated with each site.
2. 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.
3. 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 such predicate profiles. This information is aggregated across multiple runs of a program in the form of feedback report. The feedback report of each execution is a bit vector, with two bits for each predicate (observed and true), and one final bit indicating execution success or failure.

Using these reports, CBI assigns a score to every available predicate and identifies 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). In other words, a sensitive predictor is true in most failed runs but could also be true in many successful runs. On the other hand, a specific predictor is true in few successful runs but could also be true in few failed runs. Thus, a predicate is designated to be a good predictor if it is true in most failed runs and very few successful runs. CBI's scoring model considers both sensitivity and specificity to select top predictors. The best predictor is expected to point to a software bug that is responsible for many observed software 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.

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, “[i] instr; [j]?” will be used to denote 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.

3. CCI DESIGN

The ultimate goal of CCI is to automatically identify which instructions are involved in a software failure caused by concurrency bugs, such as races and atomicity violations. This section describes

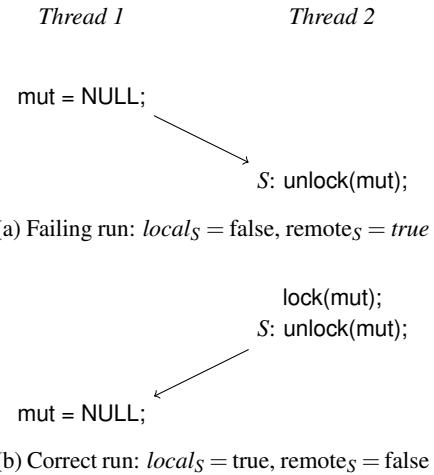


Figure 2: State of CCI predicates in two different thread interleavings. Above code is simplified from a data race bug from PBZIP2 in which thread 1 nullifies the shared mutex variable, mut, when thread 2 is still using mut.

the predicates that CCI uses, how CCI monitors these predicates at run time, and how CCI conducts sampling.

Once run-time data collection is complete, CCI uses statistical models to identify strong failure predictors among the collected data. 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.

3.1 Predicate Design

CCI instrumentation balances failure-predictive capability against profiling complexity. We do not attempt to gather anything approaching a complete trace, but rather collect just a small amount of potentially-informative data that is readily available with minimal overhead. In particular, we track whether two successive accesses to a given location were by two distinct threads or were by the same thread both times.

CCI 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 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. In the example shown in Figure 2, CCI records that $remotes_S$ is true (and $locals_S$ is false) when the execution follows Figure 2a. Conversely, CCI records that $locals_S$ is true (and $remotes_S$ is false) when the interleaving is like that in Figure 2b.

The above CCI predicates are closely tied to the root causes of concurrency bugs (e.g., atomicity violations and data races). Most atomicity violation bugs happen when one thread's consecutive accesses to some shared variable are non-serializably interleaved with accesses from a different thread [17, 27]. The Apache bug shown in Figure 1 is an example of this. Such interleavings can be captured by our $remote_I$ predicates. Data races occur when conflicting accesses from different threads touch the same shared variable without proper synchronization. The PBZIP2 bug shown in

```

1 lock(glock);
2 record(s, gTid != curTid);
3 gTid = curTid;
4 access(g);
5 unlock(glock);

```

Figure 3: Basic instrumentation

Figure 2 is a typical example of a data race bug, and likewise can be recognized using our predicate design.

Our CCI predicates are definitely not equivalent to atomicity violation predicates or race predicates. Extra information, including memory access type (i.e., read vs. write) and synchronization (i.e., when and which locks are acquired and released), is needed to precisely profile atomicity violation and data races. We choose to ignore this information and keep CCI predicates simple for following reasons:

1. Performance. Simple predicates allow more intensive sampling during production runs.
2. Generality. Our simple predicates can cover more than one type of concurrency bug, and are intentionally ignorant of the program’s synchronization mechanisms.
3. Accuracy. Though simple, our predicates do not cause excessive false positives, because CBI’s statistical models automatically prune predicates that do not explain actual, observed software failures.

3.2 Predicate Collection

As in previous CBI tools, we insert instrumentation at compile time to record predicate states when the code runs. This instrumentation is most-readily described as a source-to-source transformation of C code. We first present a basic scheme for static global variables only, then offer an enhanced scheme that considers accesses to shared memory locations across possibly-aliased pointers.

3.2.1 Basic Scheme

Our basic scheme maintains a global variable *gTid* for each global variable *g*, representing the latest thread that accesses *g*. We compare the current thread ID with *gTid* before every access to *g*, then update *gTid* with the current thread ID to set up for the next such comparison. In this manner we can easily record the state of *local_I* and *remote_I* at each instruction *I* that accesses *g*.

Figure 3 shows the detailed instrumentation created by our basic scheme. *access(g)* represents the original program instruction that accesses the global variable *g*. The thread-local variable *curTid* contains the thread id of the currently executing thread, while the variable *gTid* stores the ID of the last thread that accessed the global *g*. At line 2, we record the value of the expression *gTid != curTid* at the site *s*; *remote_I* is true and *local_I* is false if and only if this expression is true. At line 3, the value of *gTid* is updated and then the original program instruction is executed. Note that a per-global-variable lock *glock* is used to avoid races within our instrumentation code and ensure the atomicity of the whole instrumented code block.

3.2.2 Handling Pointers

The previous basic scheme assumes that we know which global variables are shared and which instructions access these variables. In practice, performing this analysis is expensive. Furthermore, a single instruction might access multiple global variables via pointers. This issue motivates us to use a completely dynamic approach to disambiguating memory addresses.

```

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

```

Figure 4: Instrumentation for pointers

We use a hash table that stores mappings from memory location addresses to the ID of the thread that last accessed each memory location. Figure 4 shows instrumentation code that uses this hash table. In practice, a single instruction might access multiple possibly-shared locations, and thereby require multiple hash table inserts and look-ups. We restrict our examples to single-access instructions for clarity of presentation. The function *test_and_insert* on line 2 sets the local variable *differs* to true if the entry in the hash table corresponding to *&g* does not equal *curTid*, and to false otherwise. Thus, the value of *differs* is true if the last thread which accessed *g* was different from the current thread executing. Also, the function inserts the entry $\&g \mapsto \text{curTid}$ into the hash table. Subsequently, the value of *differs* is recorded as shown on line 3. Also note that in this scheme the variable *glock* is a global lock that is used to control accesses to the entire hash table and ensure that the instrumented code appears to execute atomically.

In our implementation, we use a fixed size hash table in which the older entries are replaced with newer ones once the hash table becomes full. We plan to investigate other strategies to maintain this information.

3.3 CCI Sampling

As mentioned earlier, the sampling mechanism used in CBI is critical to achieving low monitoring overhead that is demanded by production run usage. Likewise, CCI randomly decides which code regions to sample at run time. The sampling rate can be adjusted to control the imposed overhead.

Unfortunately, CBI’s sampling scheme cannot be directly applied to CCI. In CBI, each thread makes local, independent decisions about when to start/stop the sampling. This is by design, and it works well for CBI’s predicates, which concern themselves with data and control activity within single threads. However, this is unsuitable for CCI: it would allow a non-sampling thread to “sneak in” and access shared data without notifying other sampling threads that it had done so. Therefore in CCI, we have to activate sampling at roughly the same time in all threads in order to collect accurate interleaving information.

To address this challenge, 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 5 shows how the basic instrumentation is augmented with sampling. If sampling is turned off, then lines 12–13 are executed. CCI uses the basic random sampling framework of CBI to set *gsample* at line 13 to turn on sampling. Once sampling is turned on the instrumentation code at lines 2–10 is enabled in all threads.

When to stop sampling is not a critical decision, provided that sampling lasts long enough time to collect useful interleaving information. In our current instrumentation scheme, the thread that sets *gsample* is the one that resets it. This is controlled by the thread-local variable *iset*, which is set at line 13 along with *gsample*. Each thread has a private copy of *iset*, so the shared *gsample* flag is cleared at the next instrumentation site in the same thread that set it in the first place or at the return of the function that set it (which is

```

1 if (gsample == 1) {
2   lock(glock);
3   test_and_insert(&g, curTid, &differs, &stale);
4   record(stale == 1 ? s1 : s2, differs );
5   access(g);
6   unlock(glock);
7   if (iset == 1) {
8     clear();
9     gsample = iset = 0;
10  }
11 } else {
12   access(g);
13   [[ gsample = iset = 1; ]];
14 }

```

Figure 5: Instrumentation with sampling. “[...]?” marks a block of code that is randomly sampled using traditional CBI sampling methods.

not shown in Figure 5). 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 [10]. We plan to explore these alternatives in the future.

An interesting issue raised by the sampling mechanism is how to update the predicates of the instruction which first accesses a memory location during a sampling period. The entry in the hash table corresponding to this memory location might not reflect the last thread which accessed it. One approach could be to not record predicates corresponding to such memory locations. The problem with this approach is that accesses to certain memory locations might be few and far between. Predicates involving successive accesses to such memory locations might never be recorded. To handle such cases, for each instruction we maintain a secondary instrumentation site which uses the stale entry in the hash table. The 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.

At the end of every sampling period, we call the function `clear()` as shown on line 8 which has the effect of marking all current entries in the hash table as being stale. This is achieved by associating a generation count with the hash table and with each entry in the hash table. An entry in the hash table is deemed stale if its generation count is less than the generation count of the hash table. When an entry is inserted its generation count is set to the current generation count of the hash table. The `clear()` function simply increments the generation count of the hash table and, hence, marking all current entries in the hash table as stale. Now, the function `test_and_insert()` on line 3 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. In this way, CCI uses information from both the current and previous sampling periods.

4. EXPERIMENTAL EVALUATION

4.1 Methodology

We have implemented CCI as an extension to the CBI framework and evaluated it using several widely used applications with real concurrency bugs. Table 1 shows some characteristics of the benchmarks and the experimental runs. The experiments were carried out to answer two key questions: (1) how accurate is CCI in report-

Table 1: Benchmark characteristics and overheads

Benchmark	Sites	Runs		Overhead	
		Total	Failed	No Sampling	Sampling
Apache	8,540	1,000	112	25%	2%
PBZIP2	420	2,000	979	200%	7%
FFT	223	1,000	322	650%	25%
LU	280	1,000	485	1,300%	800%

Table 2: Bug predictors for Apache HTTP Server

Thermometer	Predicate	Function
	R: buf->outcnt += len;	ap_buffered_log_writer()

ing root causes of concurrent program failure, and (2) what is the performance overhead of CCI monitoring? To assess the effectiveness of CCI sampling, we also compare the performance of CCI with and without sampling. All experiments were run on dual-core Intel P4 machines using a sampling deployment as recommended by earlier work [15]. We use the iterative ranking and elimination model of Liblit et al. [16] 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, also as in prior work.

Table 1 also summarizes the runtime overhead of CCI without and with sampling as compared to the uninstrumented code. Tables 2–5 visualize analysis results using *bug thermometers*, one per predictor selected by the statistical model [16]. 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.

In addition to the thread-interleaving instrumentation scheme proposed here, two conventional CBI instrumentation schemes were activated: one that records the directions of conditional branches, and one that monitors the relative values of same-typed pairs of scalar variables. Neither conventional scheme was able to diagnose failures in any of the benchmarks used: all conventional CBI predicates were eliminated due to low (< 95%) confidence that they behave differently in failing versus successful runs. This affirms our earlier claim that conventional CBI instrumentation and sampling strategies are ill-suited to diagnose concurrency bugs.

4.2 Apache HTTP Server

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 on Apache-Bugzilla [26]. Our experiments are set up based on the bug report. The whole experiment consists of 1,000 runs. 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 its log file is corrupted and a success otherwise.

Table 3: Bug predictors for PBZIP2

Thermometer	Predicate	Function
	R: pthread_mutex_unlock(fifo->mut);	consumer_decompress()

Table 2 shows all bug predictors reported by CCI, where the R indicates that the remote predicate was true at the instruction, viz., the previous access to one of the memory locations accessed was by a different thread. CCI perfectly identifies the root cause of this failure: the one listed predictor is exactly the line of code that causes failure if a different thread intervenes. By contrast, and as mentioned above, none of the conventional single-thread-oriented CBI instrumentation schemes offers a strong predictor for this bug.

Sampling is highly effective in keeping instrumentation overheads low. Without sampling, we observe a 25% slowdown. With 1/100 sampling, we find a mere 2% performance overhead, certainly suitable for use with production runs, and yet still sufficient to reveal the critical failure predictor.

4.3 PBZIP2

We also use CCI to diagnose a non-deterministic crash problem in PBZIP2 v0.9.5. This bug was originally mentioned in the change log of PBZIP2 v1.0.2. Based on the change log, PBZIP2 randomly crashes during file decompression. Our experiments are set up following the description in the change log. In addition, in order to encourage the buggy behavior to manifest more frequently, we uniformly added many thread yield calls in the source code, which were randomly executed. Finally, since CCI’s underlying source code instrumentation framework only works on C files, we ported PBZIP2 from C++. This simply required implementing the vector data structure in C, as PBZIP2 uses no other special C++ features.

Figure 2 shows a simplified version of this bug. The problem arises because the parent thread does not call `pthread_join` on the worker thread in the case of decompression. The parent then destroys the `fifo->mut` mutex. Subsequently, when the worker tries to use this mutex, the program crashes. Our experiment consists of 2,000 runs in which PBZIP2 is used to compress and decompress a file. If PBZIP2 crashes, then that run is labeled as a failure.

Table 3 shows all predictors reported by CCI. The predictor corresponds to the use of the mutex `fifo->mut` in the decompression function: precisely the location of the failure-causing race. Without sampling, CCI instrumentation causes a 3× slowdown, but 1/100 sampling reveals the bug with only a 7% performance penalty.

4.4 SPLASH-2

In this experiment, we applied CCI to two kernel programs, LU and FFT, from the SPLASH-2 benchmark suite [28]. This problem is similar to the PBZIP2 bug, i.e., the bug is caused by the bad, actually missing, implementation of the `WAIT_FOR_END` macro in the `c.m4.ia32` file used. The missing macro allows a race between the assignments at the end of the worker thread to global variables that maintain the timing information and the printing of these global variables at the end of the parent thread. As a result, the displayed timing information is incorrect.

The experiment consists of running the given program 1,000 times and marking a particular run as a failure if the finish and initialization times printed were zero. Here again, we added randomly-executed thread yield calls in the source code in order to make the program fail more frequently.

Table 4 lists all the predictors reported by CCI for FFT. The top two predictors correspond to the assignments at the end of the worker thread which store the timing information. The third predicate states

Table 4: Bug predictors for FFT

Thermometer	Predicate	Function
	R: Global->finishtime=finish;	SlaveStart()
	R: Global->initdonetime=initdone;	SlaveStart()
	R: printf("...","Global->transtimes[0,...]");	main()
	L: malloc(2*(rootN-1)*sizeof(double));	SlaveStart()

Table 5: Bug predictors for LU

Thermometer	Predicate	Function
	R: Global->rf=myrf;	OneSolve()
	L: (Global->start).gsense=-lsense;	OneSolve()

that the thread which last accessed `Global->transtimes[0]` was not the main thread. Since, in the code, only the thread which finally sets the timing statistics accesses `Global->transtimes[0]`, we see that the third predictor relates to a necessary condition for the bug to manifest, viz., the thread which sets the timing information is not the same as the main thread. The final predicate is a false positive. Note that the L indicates that the local predicate was true at the instruction. Without sampling, we see that CCI causes a 6.5× slowdown, while sampling reduces the overhead to only 25%.

Table 5 shows the predictors reported for LU. The top predictor corresponds to the assignment at the end of the worker thread which stores the timing information. For LU, without sampling, CCI instrumentation causes a 13× slowdown, and even with sampling incurs a 8× performance penalty. We are currently investigating the reason for this large overhead in this particular case.

5 RELATED WORK

Pre-deployment tools for detecting races and atomicity violations fall into two categories: static and dynamic. Static approaches [5, 9, 12] 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 [6, 17, 23] have been proposed to detect data races and atomicity violation bugs. These tools have high run-time overheads (around 25× on average) which makes them impractical for post-deployment use. Furthermore, each of these tools target only a specific class of concurrency bugs viz. either atomicity violations [6, 17] or data races [23], and assume a particular synchronization mechanism, e.g., the lockset analysis used in Eraser [23] 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 violation bugs and other types of concurrency bugs and is agnostic to which particular synchronization mechanism is used. By leveraging information about which runs failed and which were successful, CCI avoids the false-positive problems caused by benign races which plague other dynamic approaches.

These issues also apply to a recent approach by Marino et al. [18] which used happens-before relations in order to detect data races. The difference between this and previous approaches was the use of sampling to reduce overheads. Since the approach of Marino et al. is to record *all* synchronization operations for race detection, sampling can only be performed on non-synchronization memory operations. This results in run-time overheads of as much as 2.5×

for synchronization intensive applications. The sampling in CCI, on the other hand, is not constrained in this manner and can achieve much lower run-time overheads (around 10%).

There have been several other approaches to help concurrent software development. There has been a lot of research on testing of concurrent programs, such as testing based on random execution interleavings [25], exhaustive context-bounded testing [19], and race-directed random testing [24]. Record-and-replay systems 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 [4, 13]) or reliance on non-existing hardware [11]. Recent approaches [21, 22] aim to not detect, but tolerate certain kinds of races at run-time. This is orthogonal to our goal of diagnosing the root causes of concurrent software failures.

CCI is based on the Cooperative Bug Isolation (CBI) project [14, 16], 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 (sequential) bugs [2, 3].

6. CONCLUSION

We have described CCI, a low-overhead, scalable strategy for root-cause analysis of concurrency bugs. We have implemented the system and shown our technique to be effective on two widely-deployed applications. Our approach intentionally tracks far less information than exhaustive dynamic race detectors. Combined with a novel approach 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, and by experimenting with other thread-aware sampling mechanisms.

7. REFERENCES

- [1] The Apache Software Foundation. Apache HTTP Server Project. <http://httpd.apache.org/>.
- [2] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In S. Elbaum, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, London, United Kingdom, July 9–12 2007.
- [3] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2009.
- [4] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.
- [5] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.
- [6] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
- [7] J. Gilchrist. PBZIP2: Parallel BZIP2 Data Compression Software. <http://compression.ca/pbzip2/>.
- [8] P. Godefroid and N. Nagappan. Concurrency at Microsoft – an exploratory survey. TechReport, MSR-TR-2008-75.
- [9] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
- [10] 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*, pages 117–126, 2001.
- [11] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *CACM*, 2009.
- [12] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.
- [13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4), 1987.
- [14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
- [15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of Cooperative Bug Isolation. In A. Orso and A. Porter, editors, *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 57–62, Edinburgh, Scotland, May 24 2004.
- [16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
- [17] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.
- [18] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.
- [19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *PLDI*, 42(6):446–455, 2007.
- [20] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.
- [21] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in concurrent programs. In *ASPLOS*, 2009.
- [22] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP*, 2009.
- [23] 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.
- [24] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
- [25] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.
- [26] A. Sussman and J. Trawick. Corrupt log lines at high volumes. https://issues.apache.org/bugzilla/show_bug.cgi?id=25520.
- [27] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.
- [28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.