Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments

Brandon Lucia Benjamin P. Wood Luis Ceze

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

Abstract

In this paper we propose *Recon*, a new general approach to concurrency debugging. Recon goes beyond just detecting bugs, it also presents to the programmer short fragments of buggy execution schedules that *illustrate how and why bugs happened*. These fragments, called *reconstructions*, are inferred from inter-thread communication surrounding the root cause of a bug and significantly simplify the process of *understanding* bugs.

The key idea in Recon is to monitor executions and build graphs that encode inter-thread communication with enough context information to build reconstructions. Recon leverages reconstructions built from multiple application executions and uses machine learning to identify which ones illustrate the root cause of a bug. Recon's approach is *general* because it does not rely on heuristics specific to any type of bug, application, or programming-model. Therefore, it is able to deal with single- and multiple-variable concurrency bugs regardless of their type (*e.g.*, atomicity violation, ordering, etc). To make graph collection efficient, Recon employs selective monitoring and allows metadata information to be imprecise without compromising accuracy. With these optimizations, Recon's graph collection imposes overheads typically between 5x and 20x for both C/C++ and Java programs, with overheads as low as 13% in our experiments. We evaluate Recon with buggy applications, and show it produces reconstructions that include all code points involved in bugs' causes, and presents them in an accurate order. We include a case study of understanding and fixing a previously unresolved bug to showcase Recon's effectiveness.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Debugging Aids

General Terms Algorithms, Reliability

Keywords concurrency, debugging, atomicity violation, order violation, multi-variable, data race, concurrency bug, communication graph

1. Introduction

Concurrency bugs are a major stumbling block to writing multi-threaded programs. Even expert programmers puzzle over complicated behaviors resulting from the unexpected interaction of operations in different threads. Developers face errors such as data races, atomicity violations, deadlocks, and ordering errors.

Concurrency bugs are particularly difficult to diagnose and fix for two main reasons. First, developers must reason about the interactions of many pieces of code executing in multiple threads. Observing one thread's buggy behavior is often insufficient for understanding the cause of the bug, which may lie in another thread. Second, non-determinism in multi-threaded execution complicates the process of interpreting buggy executions. The exact behavior of the application may vary, even from one buggy run to the next, making it very difficult to pin-point the root cause of the bug. Prior work [16] showed that the challenges of concurrency have led developers to give up on some bugs entirely, or to apply incorrect or incomplete stop-gap solutions. As a result, errors remain in the wild, potentially leading to surprising software failures.

There is a large body of work addressing concurrency bugs. A significant fraction of prior work focuses on dynamically detecting data races [6, 25], atomicity violations [8, 17], and locking or sharing discipline violations [1, 26, 33]. Some recent work on testing investigated new ways of systematically exploring executions [5, 21] and replaying buggy executions [24, 31]. Prior work has had considerable success, but there still remains much to be addressed.

First, many prior approaches to *detect* bugs report *too little* information to understand bugs: a single communication event [18, 27, 32] or the thread preemptions from buggy runs [5, 21]. However, concurrency bugs are complex and involve code points distributed over a code base and in multiple threads, requiring more information to be fully understood. To *understand* such bugs, developers benefit from seeing a portion of the execution illustrating the actual code interleaving that led to buggy behavior. Second, replay-based approaches [24, 31] often report *too much* information — effectively, the entire execution schedule. Replay makes bugs reproducible, but programmers must sift through an entire execution trace to comprehend bugs. Finally, many techniques are tailored to a specific class of concurrency errors [17, 23], limiting their applicability. It is infeasible to anticipate every possible error scenario, and design a tool targeting each. Hence, *generality* is crucial.

We propose Recon, a new approach to concurrency debugging based on *reconstructions* of buggy executions. Reconstructions are short, focused fragments of the interleaving schedule surrounding a program event, such as shared-memory communication. Figure 1 illustrates what a reconstruction is and how a reconstruction relates to an execution. Observe that not all program events are included in the reconstruction. Instead, a reconstruction contains a concise summary of program behavior surrounding an event that is likely to be related to a concurrency error. Reconstructions are based on communication graphs that encode information about the ordering of communication events. Based on this ordering, reconstructions show the interleaving that caused buggy behavior, rather than just some of the code points involved. Reconstructions are *general*, as they make no assumptions about the nature of bugs — *i.e.*, Recon does not look for specific patterns.

Figure 2 shows an overview of Recon's basic operation. The process begins when a programmer observes a bug or receives a bug report. The programmer then derives a test case designed to trigger the bug, and runs the test multiple times under Recon. Recon collects a communication graph from each execution, and the programmer or test environment labels each graph as buggy or nonbuggy, depending on the outcome of the test. Recon then builds reconstructions from edges in buggy graphs; for each one, Recon computes statistical features to quantify the likelihood that they are related to the bug. Recon uses the features to compute a rank for each reconstruction, and presents them to the user in rank order.

With Recon, we make several contributions:

- We propose the concept of reconstructing fragments of multi-threaded executions and develop an algorithm that builds reconstructions from communication graphs.
- We propose a set of features to describe reconstructions and use statistical techniques to identify reconstructions that illustrate the root cause of bugs.
- We develop optimization techniques to build communication graphs efficiently.
- We implement Recon for both C/C++ and Java. Our evaluation uses bugs from the literature, including several large applications, and shows that Recon precisely identifies bugs and their corresponding reconstructions. We include a case study of using Recon to understand and fix an unresolved bug.

The rest of this paper is organized as follows. Section 2 discusses concurrency bugs in general and provides an overview of the role of communication graphs for debugging. Section 3 discusses our approach to reconstructing execution fragments. Section 4 describes how we use machine learning techniques to identify bugs. Section 5 describes our implementation of Recon for C/C++ and Java, and several enabling optimizations. Section 6 describes our experiments, which show that Recon reconstructs buggy executions precisely and efficiently. Section 7 discusses related work and Section 8 concludes.

2. Background

2.1 Concurrency Bugs

Data races occur when two different threads access the same memory location, at least one access is a write, and the accesses are not ordered by synchronization. *Ordering violations* involve two

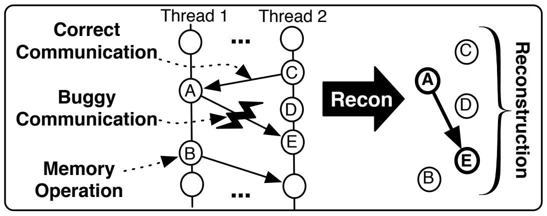
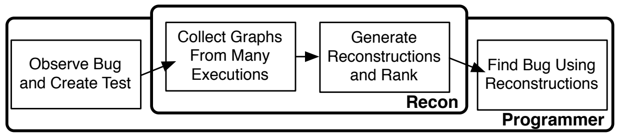


Figure 1. Recon reconstructs fragments of program execution.



Observe Bug and Create Test
Collect Graphs From Many Executions
Generate Reconstructions and Rank
Recon
Find Bug Using Reconstructions
Programmer

Figure 2. Overview of Recon's operation.

or more memory accesses from multiple threads that happen in an unexpected order, due to absent or incorrect synchronization.

Atomicity violations result from a lack of constraints on the interleaving of operations in a program. Figure 3(a) illustrates an atomicity violation bug that was found in `weblech`, a multi-threaded web crawler. The program uses a shared queue that was implemented incorrectly. The check of the queue's size on line 168 should be atomic with the dequeue operation performed on line 189 to ensure that there is always an item to dequeue, but the programmer has not implemented this atomicity constraint.

Figure 3(b) shows an execution trace manifesting the bug. In this trace, the `size()` and `dequeue()` calls in Thread 2 interleave between the `size()` and `dequeue()` calls in Thread 1. Since the queue is emptied by Thread 2, Thread 1's call to `dequeue()` returns `null`, which is stored in the local variable `item`. Thread 1 later crashes with a `NullPointerException` when trying to invoke the `getD()` method on this null `item`.

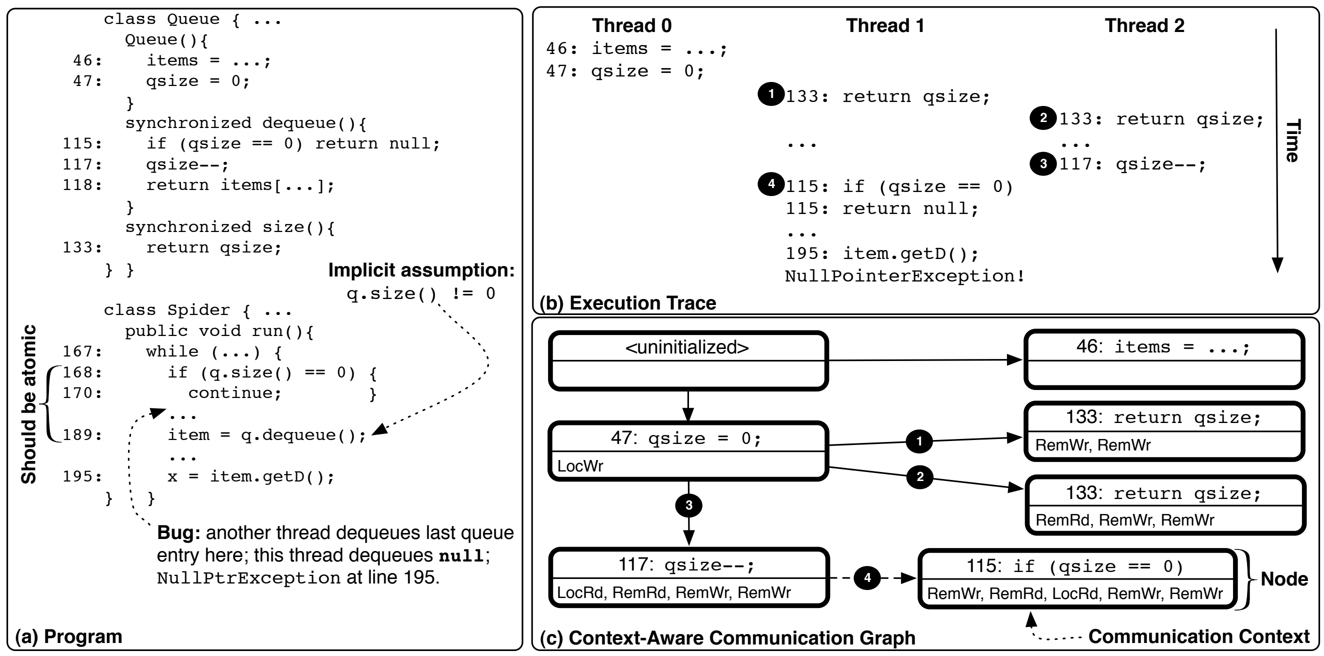
2.2 Communication Graph Debugging

Prior work [18, 27, 32] has observed that *communication graphs* are useful for detecting concurrency errors. A communication graph is a representation of a program execution that captures inter-thread data-flow. A node in a communication graph represents a memory instruction at some program point. An edge between two nodes, the *source* and *sink*, indicates that the sink instruction read or overwrote data written by the source instruction. Furthermore, the source and sink instructions executed in different threads.

The main idea behind concurrency debugging with communication graphs is that a problematic communication event characterizes the error's behavior, and is represented by an edge in the graph. Identifying the problematic communication event illustrates the bug's behavior to a developer, aiding in debugging. An approach used in prior work for identifying problematic communication is to focus on communication events that tend to occur often in buggy program runs, and infrequently or never in correct program runs [17, 32]. This technique is often useful, but insufficient in general.

To understand why, refer back to the example in Figure 3(b). The problematic communication event is Thread 1's read of the queue's `qsize` field on line 115; it reads a value written by Thread 2 at line 117 rather than the same value Thread 1 read from `qsize`, at line 133. Looking at this communication alone is insufficient to find the bug; the involved instructions communicate in both buggy *and* nonbuggy executions. This difficulty in identifying the problematic edge in a communication graph was the motivation for developing *context-aware communication graphs* in Bugaboo [18].

Context-Aware Communication Graphs. Bugaboo first demonstrated that communication graphs are insufficient for general concurrency bug detection. Bugaboo addresses this inadequacy by adding *communication context* to communcation graphs. Communication context is a short (*e.g.*, 5 entries) history of *context events* that is maintained by each thread in an execution. Context events are *Local Reads*, *Local Writes*, *Remote Reads*, and *Remote Writes*. A thread records a "Local" event in its context for sharing read



```
         class Queue { ...
           Queue(){
   46:       items = ...;
   47:       qsize = 0;
           }
           synchronized dequeue(){
  115:       if (qsize == 0) return null;
  117:       qsize--;
  118:       return items[...];
           }
           synchronized size(){
  133:       return qsize;
         } }

         class Spider { ...
           public void run(){
  167:       while (...) {
  168:         if (q.size() == 0) {
  170:           continue;        }
               ...
  189:         item = q.dequeue();
               ...
  195:         x = item.getD();
         } }
```

```
Thread 0                   Thread 1                     Thread 2
46: items = ...;
47: qsize = 0;
                        1 133: return qsize;
                                                     2 133: return qsize;
                          ...                          ...
                                                     3 117: qsize--;
                        4 115: if (qsize == 0)
                          115: return null;
                          ...
                          195: item.getD();
                          NullPointerException!
```

Figure 3. Example showing (a) a buggy program, (b) a bug-triggering execution trace, and (c) the context-aware communication graph produced by the trace. Nodes represent the execution of operations in a specific context. Edges represent communication between nodes. Note that we only include events in nodes' contexts that appear in our abbreviated trace for presentation purposes.

and write operations that it executes. A thread records a "Remote" event in its context when another thread performs a read or a write to a memory location that it has accessed recently. Context events represent memory operations performed by any instruction, to any address, and only events' types (*e.g.*,"Remote Read") are added to a thread's context.

Communication context is the basis for context-aware communication graphs. In a context-aware graph a node is a pair (I, C) representing the execution of a static instruction, I, in communication context, C, instead of just an instruction. Edges in a context-aware graph represent communication between dynamic instruction *instances*. In a context-aware graph, there may be more than one node per static instruction, but the size of the graph is bounded by a function of the context size, as described in prior work [18].

Figure 3(c) shows a context-aware communication graph produced by the execution trace in Figure 3(b). The numbered circles map events in the trace to their corresponding edge in the graph. In the context-aware graph, the sink node of edge 4 occurs only in buggy executions. The most recent (left-most) two events in this node's context are the Remote Write that corresponds to Thread 2's write at line 117, and the Remote Read that corresponds to Thread 2's read at line 133. This context reflects the buggy interleaving of Thread 1's read of `qsize` at line 133, and its read of `qsize` at line 115. Hence, edge 4, which includes this unique context-aware node, occurs in buggy executions' graphs, and not correct executions' graphs. Context-aware graphs provide a way to identify buggy communication in such complex bugs.

Debugging with communication graphs is one of the key motivating ideas behind Recon. However, rather than just isolating graph edges likely to represent problematic communication, as most prior work has done, Recon reconstructs *temporal sequences* of communication events and, using machine learning, infers which sequences most likely illustrate a bug's cause.

3. Reconstructing Execution Fragments

The goal of Recon is to simplify debugging by presenting the programmer with a short, focused, temporally ordered reconstruction of the events that were responsible for buggy behavior. Our technique for reconstructing execution fragments is based on a specialized version of context-aware communication graphs [18].

3.1 Timestamped Communication Graphs

We specialize the context-aware communication graph abstraction to encode ordering between non-communicating nodes. We do so by adding a timestamp to each node, indicating when the node's instruction executed in the node's context. We call this new graph abstraction the *timestamped communication graph*; referring to them as just "graphs" hereafter. To summarize the structure of the graphs used by Recon:

- A node is a pair (I, C) representing the execution of static instruction I in communication context C. Each node is labeled with a timestamp, T, representing the global time when instruction I last executed in context C.
- An edge is a pair of nodes (u, v) representing communication from the instruction instance represented by u to the instruction instance represented by v.

Graph Construction. We collect graphs by keeping a *last-writer record* for each memory location, storing: (1) the thread that last wrote the location; (2) the instruction address and context of that write; and (3) a timestamp for the access. When a memory location is accessed by a different thread than the thread that last wrote the location, an edge is added to the graph. The edge's source node is populated with the instruction address, context, and timestamp from the location's last-writer record. The edge's sink node is populated with the instruction address, context, and timestamp of the operation being performed.

To limit the size of the graph, we record only the most recent pair of timestamps for an edge, *i.e.*, the timestamp is not used to identify a node, only the instruction and context are. When adding a communication edge to the graph, if the edge already exists, only the timestamps are updated. By overwriting timestamps, we lose some ordering information, so we call our extension a *lossy timestamp*. Figure 4(a) shows an example of a timestamped communication graph. The graph is similar to the one in Figure 3, except that each node now has a timestamp indicating when it last occurred.

3.2 Reconstructions

A *reconstruction* is a schedule of communicating memory operations that occurred during a short fragment of an execution. In this section, we describe the process of building a reconstruction around a single, arbitrary communication event (*i.e.*, graph edge). Section 4 describes how we identify the reconstructions most likely related to bugs, and Section 5.3 details the entire debugging process.

3.2.1 Building Reconstructions from Graphs

Recon builds reconstructions starting from an edge in a graph. In addition to the instructions represented by the source and sink nodes of the edge, a reconstruction should include the memory operations that executed in a short window prior to the source node, in the time between the source and the sink nodes, and in a short window following the sink. These *regions* of the execution are called the *prefix*, *body*, and *suffix* of the reconstruction, respectively, and are selected from the graph for a single execution according to nodes' timestamps.

The size of the window of nodes considered in computing the prefix and suffix is arbitrary. With a larger window, there is a greater chance that unrelated nodes are included in a reconstruction. With a smaller window, fewer unrelated nodes are likely to end up in a reconstruction, but we risk excluding events related to the bug that occur far away from the communication event. A reasonable window size heuristic is to use the length of the communication context of a node. Using the context length, we include nodes that were influenced by, or influenced the context of the sink or source.

Formal Definition of Reconstruction A reconstruction is a tuple (e, p, b, s). The reconstruction is built around an edge, e, with source node, e_{src} and sink node, e_{sink}. The prefix, p, is a set of consecutive nodes immediately preceding e_{src} in timestamp order. The body, b, is the set of all nodes between e_{src} and e_{sink} in timestamp order. The suffix s is a set of consecutive nodes immediately following e_{sink} in timestamp order. The cardinalities of the prefix and suffix are bounded by fixed constants. The cardinality of the body is bounded by a threshold function that we describe in Section 5.3.

3.2.2 Simpler Debugging Using Reconstructions

Figure 4(b) shows the reconstruction Recon produces from one of the edges in the graph in Figure 4(a). This reconstruction illustrates the buggy interleaving of queue operations shown in Figure 3(b). It includes all the code points involved in the bug, and presents them in the order that leads to buggy behavior. In buggy runs, the read of the queue's size on line 133 and the dequeue on line 115 are interleaved by the dequeueing decrement of `qsize` at line 117. This buggy interleaving is clear in the reconstruction: line 133's node is in the body and line 115's node is in the suffix. The interleaving dequeue operation at line 117 is the sink node, ordered between the body and the suffix.

This example illustrates a key contribution of reconstructions. Looking at the buggy edge between line 117's node and line 115's node does not explicitly indicate the bug — it suggests the involvement of the queue, but not the atomicity violation involving line 133. Instead, the reconstruction built around the *nonbuggy* edge from line 47's node and line 117's node illustrates the bug, showing all three involved code points *and* the buggy execution order.

3.3 Aggregate Reconstructions

We have described how to build a reconstruction for a single edge from a single execution. We can aggregate reconstructions from a *set* of runs to see how frequently code points occur in a region of a reconstruction *across* executions. This information allows us to define our confidence that a code point belongs in a region.

Starting from a set of graphs, we compute a reconstruction for each edge in each execution's graph individually. We then aggregate the reconstructions of each edge across the executions by computing the union of each of their prefixes, the union of each of their bodies, and the union of each of their suffixes, producing the aggregate prefix, body, and suffix. A node may occur in multiple different regions in an aggregate reconstruction, if, for instance, in half of executions it appeared in the prefix, and in half it appeared in the body. Nodes in the same region in an aggregate reconstruction are unordered with one another, but are ordered with the source and the sink of the edge in the reconstruction, and with nodes in other regions. Nodes within a region are unordered because timestamps are not comparable across executions.

When aggregating reconstructions, we associate a confidence value with each node in a region. The confidence value is equal to the fraction of executions in which that node appeared in that region. The confidence value of a node in a region represents the likelihood that a node occurs in that region. In Section 4, we discuss using confidence values to identify reconstructions likely related to buggy behavior.

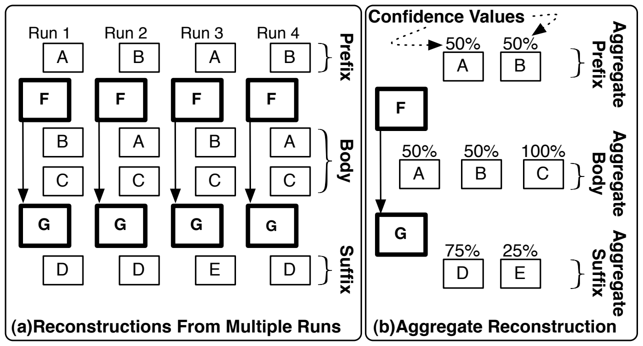


Figure 5. (a) Reconstructions of many runs and (b) the resulting aggregate reconstruction with confidence values.

Figure 5 shows an example of the aggregation process. Part (a) shows reconstructions produced from 4 different executions, and part (b) shows the aggregate reconstruction produced from these 4 reconstructions. In this example, node A appears in the prefix of half of the reconstructions, and appears in the body in half of the reconstructions. The prefix and body of the aggregate reconstruction therefore both include node A, and assign it a confidence value of 50%. Node C appears in the body of all of the individual reconstructions, so it appears in the body of the aggregate reconstruction with a confidence value of 100%.

4. Debugging with Reconstructions

There are four steps to debugging a program with Recon:

1. The program is run under Recon several times, yielding a set of timestamped communication graphs labeled buggy or nonbuggy (Section 2.1).

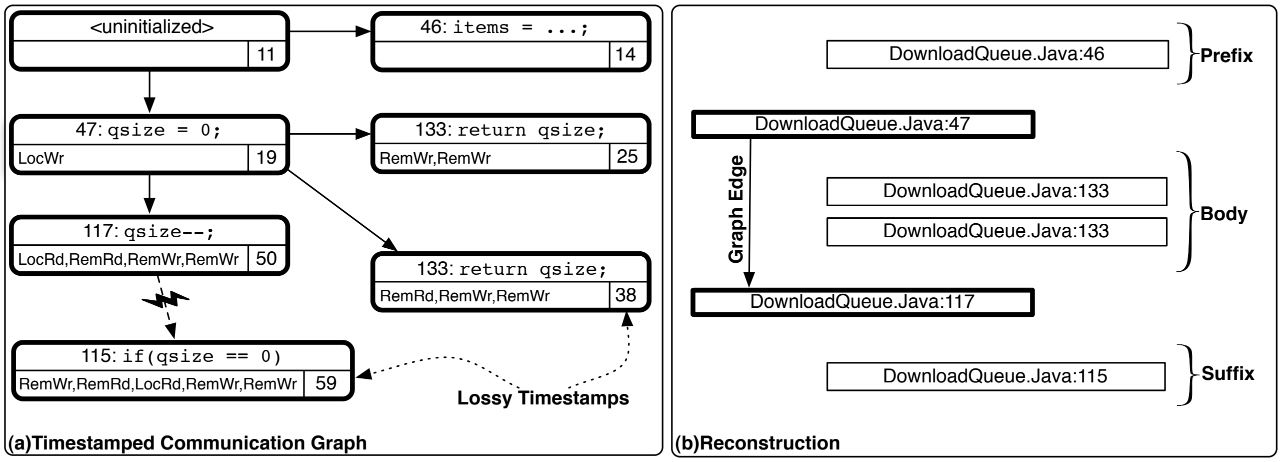


Figure 4. A timestamped communication graph (a) and reconstruction (b). The graph corresponds to Figure 3(c).

2. Next, Recon must decide which edges are worth using as the basis for a reconstruction. Recon selects edges from each buggy graph based on how strongly correlated they are with the occurrence of buggy behavior. Section 4.1.1 describes this correlation.
3. For each selected edge, Recon builds and aggregates reconstructions (Sections 3.2 and 3.3).
4. Finally, Recon ranks the reconstructions by how likely they are to illustrate the bug, as determined by a set of *features* computed from the aggregated reconstructions and the nonbuggy graphs.

We now discuss the features we developed, and how we use them to produce a reconstruction's rank.

4.1 Features of Reconstructions

A key design concern is that features are *general*. A feature that targets one bug type or pattern is not as useful. If we choose features that are not general, we bias our search toward some bugs and miss others entirely. For example, serializability analysis of memory access interleavings has been used to detect atomicity violations [17, 23]. However, it does not detect ordering bugs or any multi-variable bugs.

Our features should capture as much information as necessary to discriminate between reconstructions of buggy fragments of an execution and reconstructions of nonbuggy fragments. We use three features: *Buggy Frequency Ratio* focuses on the correlation between communication events and buggy behavior; *Context Variation Ratio* focuses on variations in communication context that correlate with buggy behavior; and *Reconstruction Consistency* focuses on the consistency with which sequences of code points occur in reconstructions from buggy executions. The rest of this section describes these features in detail and verifies their efficacy empirically using a feature importance metric from machine learning [14].

4.1.1 Buggy Frequency Ratio (B)

Intuition. A reconstruction's Buggy Frequency Ratio, or B value, describes the correlation between the frequency of the communication event from which the reconstruction was built, and the occurrence of buggy behavior. The motivation for this feature is that we are interested in events in an execution that occur often in buggy program runs, but rarely, or never, in nonbuggy runs.

Definition. For each aggregate reconstruction, assume $\#Runs_b$ buggy runs and $\#Runs_n$ nonbuggy runs. Assume the reconstruction's edge occurred in $EdgeFreq_b$ buggy runs and in $EdgeFreq_n$ nonbuggy runs. The fraction of buggy runs in which the edge occurred is:

$$Frac_b = \frac{EdgeFreq_b}{\#Runs_b}$$

The fraction of nonbuggy runs in which the edge occurred is:

$$Frac_n = \frac{EdgeFreq_n}{\#Runs_n}$$

We define B as follows:

$$B = \frac{Frac_b}{Frac_n}$$

If a reconstruction's edge occurs in many buggy runs and few nonbuggy runs, B is large. Conversely, if the edge occurs often in nonbuggy runs and rarely in buggy runs, B is small.

If the edge never occurs in a nonbuggy run, but occurs in buggy runs, then it is very likely related to the bug. However, in such a case, $Frac_n$ is 0, and unless we handle this case specially, B is undefined. In this corner case, we give $Frac_n$ a value that is smaller than the value produced if the edge occurs in one nonbuggy run (by assigning $Frac_n = \frac{1}{\#Runs_n+1}$). This yields large B values for these important edges.

4.1.2 Context Variation Ratio (C)

Intuition. The Context Variation Ratio (C) quantifies how variation of contexts of communicating code points correlates with buggy execution. We can determine the pair of communicating code points in the edge around which a reconstruction is built, since a node is identified by an instruction and context. From that, we can determine all edges involving that pair of code points, regardless of context; we then compute the set of all contexts in which the pair communicated. In a program that has frequent, varied communication, there are many contexts in this set; in a program with little — or less-varied — communication, the set is small. We consider a reconstruction suspicious if the pair of code points forming the edge around which the reconstruction was built execute in a substantially different number of contexts in buggy runs than in correct runs.

Definition. For a reconstruction built around an edge between two code points, we define $\#Ctx_b$ to be the number of contexts in which the code points communicated in buggy runs, and $\#Ctx_n$ to be the number in nonbuggy runs. C is the ratio of the absolute difference of $\#Ctx_b$ and $\#Ctx_n$ to the total number of contexts for the pair of code points in nonbuggy and buggy runs. We define C as follows:

$$C = \frac{|\#Ctx_b - \#Ctx_n|}{\#Ctx_b + \#Ctx_n}$$

Large C values indicate a disparity in communication behavior between buggy and nonbuggy runs. Hence, a reconstruction with a large C value more likely illustrates the communication pattern that led to buggy behavior.

4.1.3 Reconstruction Consistency (R)

Intuition. Reconstruction Consistency (R) is the average confidence value over all code points in an aggregate reconstruction. R is useful because code points that consistently occur in reconstructions of buggy executions are likely related to the cause of the bug. As described in Section 3.2, each node in an aggregate reconstruction has an associated confidence value that represents the frequency with which it occurs at a certain point in that reconstruction. In an aggregate reconstruction produced from a set of buggy runs, a node with a high confidence value occurs consistently in the same region of reconstructions from those buggy runs. Such nodes' operations are therefore likely to be related to the buggy behavior in those runs. Reconstructions containing many high confidence nodes reflect a correlation between the *co-occurrence* of those nodes' code points in the order shown by the reconstruction, and the occurrence of buggy behavior.

Definition. We compute R for a reconstruction as the average confidence value over all its nodes. Formally, for a reconstruction with prefix region P, body B, and suffix S and where $V(n, r)$ is the confidence value of node n in region r, we define R as follows:

$$R = \frac{\sum\limits_{p \in P} V(p, P) + \sum\limits_{b \in B} V(b, B) + \sum\limits_{s \in S} V(s, S)}{|P| + |B| + |S|}$$

Nodes in a reconstruction with a large R value tend to occur in the reconstructed order when buggy behavior occurs. Such reconstructions are therefore more likely to represent problematic interleavings, and to be useful for debugging.

4.2 Using Features to Find Bugs

By construction, large values for B, C, or R indicate that a reconstruction is likely to be buggy. Therefore, we give each reconstruction a score equal to the product of all non-zero features' values. We rank reconstructions, with highest scoring reconstruction first.

Empirical validation of B, C, and R. We now quantitatively justify our features using real buggy code (we describe our experimental setup in Section 6). We assessed the discriminatory power of our features using Weka's [9] ReliefF [14] feature selection function. The magnitude of a feature's ReliefF value is greater if the distance between points of different classes (*e.g.*buggy or non-buggy) is greater along that feature's dimension, on average. The magnitude of a feature's ReliefF value corresponds to how well it discriminates between classes.

Table 1 shows ReliefF values for bugs in several C/C++ applications. All features' ReliefF values are non-zero, meaning they are useful for classification, and many have ReliefF values close to 1.0. The relative importance of features varies by program.

For `apache`, B and R are the most useful. C is less important, indicating there is a similar amount of context variation in buggy and nonbuggy runs. Figure 6 illustrates the relative importance of the features graphically, with pair-wise feature plots. Figure 6(a) shows that when viewed along the axes of highest ReliefF, there is clear segregation of buggy and nonbuggy reconstructions. In the plot, buggy points tend to the upper right, meaning they have larger feature values than nonbuggy points. Figures 6(b) and (c) reiterate the class segregation along the B and R axes, and illustrate the less clear division along the C axis.

	ReliefF Rank		
Program	B	R	C
apache	0.99	0.91	0.16
mysql	0.20	0.59	0.76
pbzip2	0.26	0.28	0.28
aget	0.84	0.91	0.16

Table 1. ReliefF rank of features for C/C++ programs.

`pbzip2`'s ReliefF values are smaller than other applications' values. The disparity indicates that in each dimension, `pbzip2`'s buggy and nonbuggy points tend to be nearer to one another than in other applications. Hence, ranking by a single feature is inadequate to isolate bugs precisely. However, in the three-dimensional space of all features, buggy and nonbuggy reconstructions are far apart. As our results in Section 6.2 confirm, ranking using all three features isolates the reconstruction of the bug in `pbzip2`.

Our ReliefF feature analysis emphasizes two properties of our technique: (1) our features precisely classify buggy reconstructions to identify bugs; and (2) considered together, our features are more powerful than each individually.

5. Implementation

We implemented two versions of Recon: one for C/C++, using Pin [19], and one for Java, using RoadRunner [7]. The implementation has three parts: (1) tracking communication; (2) collecting graphs; and (3) generating and ranking reconstructions.

5.1 Tracking Communication

To track communication we maintain a *metadata table*. This table maps each memory location to an entry containing its last-writer record, and a list of threads that have read from the location since its last write that we call the *sharers list*. Each thread has a communication context. A thread's context is a shift register of events, as described in Section 3. We use a 5-entry context.

When a thread writes to a memory location, it updates the location's last-writer record with its thread ID, the instruction address of the write, its current context, and the current timestamp. If the writing thread is different from the last writer, it does three things: (1) update its context with a local write event; (2) update the context of each thread in the sharers list with a remote write event; and (3) clear the sharers list.

When a thread reads a location, it looks up the last writer thread in the last-writer record. If the reading thread is different from the last writer, it does three things: (1) update its context with a local read; (2) update the last writer thread's context with a remote read; and (3) add the reading thread to the memory location's sharers list.

For C/C++, we implement the metadata table as a fixed size hash table of 32 million entries. To find a memory location's metadata, we index with the address modulo the table size. We use a lossy collision resolution policy: on a hash collision, an access may read or overwrite another colliding location's metadata. We ignore stack accesses, as they are rarely involved in communication. For Java, we use RoadRunner's shadow memory to implement a distributed metadata table. Its size scales with allocated memory and it does not suffer from collisions. Unique identifiers of memory access instructions in the bytecode replace instruction addresses. Contexts are stored as integers, using bit fields. We instrument accesses to fields and arrays, but not local variables.

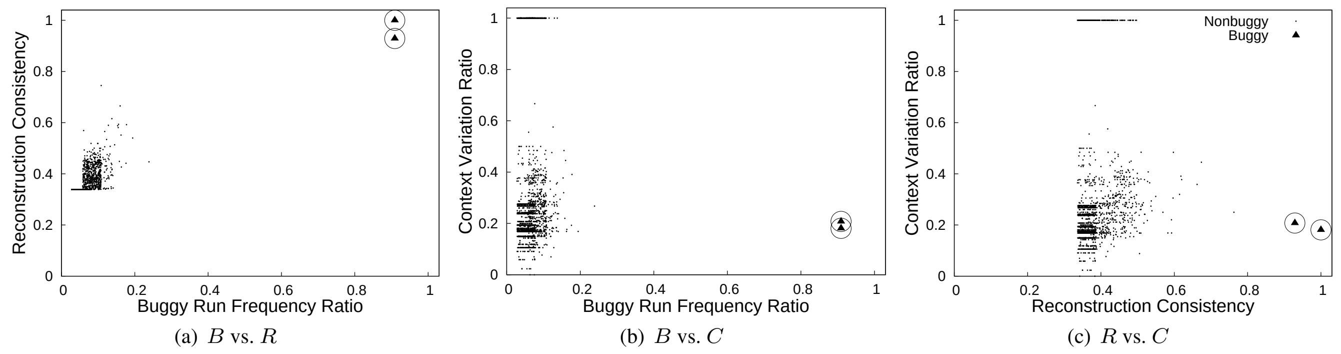


Figure 6. Pair-wise plots of features for `apache` showing the top 2000 ranked reconstructions. Buggy reconstructions' points are circled.

5.2 Timestamped Communication Graphs

Each thread maintains its own partial communication graph. Partitioning the communication graph over all threads makes adding an edge a thread-local operation, which is critical for performance. When a thread tries to add an edge, it first searches the graph for the edge. If the edge is already in the graph, the thread overwrites the existing timestamps with the timestamps of the edge being added. If not, a new edge is created. When a thread ends, it merges its partial graph into a global graph. Once all partial graphs are merged into the global graph, it is written to a file.

For C/C++, we use the `RDTSC` x86 instruction to track timestamps. We maintain communication graphs as a chaining hash table. Separately for the source and sink node, the hash function sums the entries in each node's context. Each node's sum is then XORed with the node's instruction address. The result of the computation for the source node is then XORed with the result of the computation for the sink, producing the hash key. For Java, we generate timestamps from the system time and implement communication graphs as adjacency lists.

5.3 Generating and Ranking Reconstructions

We generate and rank reconstructions with the following process. We separately load sets of buggy and nonbuggy graphs into memory and create a list of nodes ordered by timestamp for each buggy run. At this point, we compute C and B for each edge in the set of buggy graphs. We then rank these edges by their B values. Next, we generate reconstructions for the top 2000 edges ranked by B, using the algorithm described in Section 3.2. To limit the size of the reconstructions produced, we limit the number of code points in each region. To do so, we threshold by confidence value, excluding from a reconstruction any node that has a confidence value less than half the region's maximum confidence value. After computing reconstructions, we compute their R values and their ranks, and output them in rank order.

5.4 Optimizing Graph Collection

We use two optimizations to reduce overheads: (1) we reduce the number of instructions for which analysis is required and (2) we permit an instrumentation data race to avoid locking overheads.

5.4.1 Selectively Tracking Memory Operations

The simplest way of reducing graph collection overhead is monitoring fewer memory operations. We develop two optimizations to do so. They can lead to lost or spurious edges, but our results (Section 6) show that Recon's accuracy is unaffected.

First Read Only. Repeated reads to a memory location by the same thread are likely redundant. We therefore develop the *first-read* optimization: threads perform analysis only on their first read to each location after a remote write to that location. Skipping updates on subsequent reads is analogous to performing analysis only on cache read misses. Due to the frequency and temporal locality of reads, this optimization eliminates many updates.

This optimization is *lossy*. If a thread repeatedly reads the result of a write, only its first read is reported. If subsequent reads are performed by different code or in different contexts they will not cause edges to be added to the graph. Additionally, context events corresponding to ignored reads are not published to threads' contexts, which may result in fewer distinct contexts and edges.

First Write Only. Repeated writes by the same thread are often redundant or non-communicating. Under the *first-write* optimization, a thread only updates the last-writer table and sharers list on a write to a memory location x when it is not the last thread to write x. This optimization is *noisy*. If a thread that is not the last writer of x writes to x and does not update x's metadata on subsequent writes, another thread's read of x may see outdated metadata and add a spurious edge with incorrect context information to the graph.

5.4.2 Intentional Instrumentation Races

On every memory access, threads check the last writer of the location they are accessing to determine what analysis operations must be performed, as described in Section 5.1. To ensure threads observe consistent metadata, they acquire a lock on each access.

We observe, however, that due to temporal locality these checks are often performed by the location's last writer. In such situations, reading all metadata is unnecessary. The cost of acquiring the lock just to check the last writer outweighs the cost of the check itself. To mitigate this cost, we can perform the check without holding the lock, which we call the *racy-lookups* optimization. If, based on the check, a thread determines it must perform further analysis or update the metadata, it acquires the lock. Only the check to determine the location's last writer races.

In principle, data races can lead to undefined behavior [4] or memory inconsistency [20]. In practice, there are only two inconsistent outcomes of this optimization. The first is that the last writer performs a check that indicates it is not the last writer. In this case, the checking thread last wrote the metadata. On x86 the thread will correctly read its own write, making this situation impossible. In Java, our metadata writes are well ordered, and the read involved in the check is ordered with its metadata update. As a result, the checking thread can only ever correctly read that it was the last writer. The other inconsistency is when a check indicates to a thread that it is the last writer when it is not. This situation is possible in x86 and Java. On x86, the check reports that the checking thread is the last writer, so it does no analysis. Because the check

	Category	Program	Version	Bug Type
C/C++	Bug Kernel	logandswp	n/a	Atomicity Violation
		circlist	n/a	Atomicity Violation
		textreflow	Mozilla 0.9	Multi-Variable Atomicity Violation
		jsstrlen	Mozilla 0.9	Multi-Variable Atomicity Violation
	Full App.	apache	httpd 2.0.48	Atomicity Violation
		mysql	mysqld 4.0.12	Atomicity Violation
		pbzip2	pbzip2 0.9.1	Ordering Violation
		aget	aget 0.4	Multi-Variable Atomicity Violation
Java	Bug Kernel	stringbuffer	JDK 1.6	Multi-Variable Atomicity Violation
		vector	JDK 1.4	Multi-Variable Atomicity Violation
	Full App.	weblech	weblech 0.0.3	Atomicity Violation

Table 2. The buggy programs we used to evaluate Recon

was not synchronized, however, another thread's update to the last-writer field may have been performed, but not yet made visible to all threads. In this case, the checking thread should have seen the update, and added an edge, but did not. This situation can also arise because our instrumentation is not atomic with program accesses. In practice, it has little impact on our analysis. Furthermore, the statistical nature of Recon is robust to noise, so such omissions do not impact Recon's bug detection capability.

6. Evaluation

There are several components to our evaluation. We show that our ranking technique is effective at finding bugs and that the reconstructions Recon produces are useful and precise. We show that Recon requires few program runs in order to be effective. We describe a case study of our experience fixing a previously unresolved bug. Finally, we show that with our optimizations Recon's overheads are similar to other analysis tools, and overall data collection time is short.

6.1 Experimental Setup

We evaluated Recon's ability to detect concurrency bugs using the buggy programs described in Table 2. We used a set of full applications, as well as several bug kernels. Our bug kernels are shorter programs with bugs extracted from the literature (`stringbuffer`, `vector`, `circlist`, `logandswp`), and buggy sections of code extracted from full versions of the Mozilla project (`textreflow`, `jsstrlen`). Our benchmarks encompass many bug types observed in the wild [16] including ordering bugs and single- and multiple-variable atomicity bugs. We ran each application in Recon with all optimizations. Our test script used external symptoms such as crashes or corrupt output to label graphs.

We evaluated Recon's runtime and memory overhead, compared to uninstrumented execution. We used the PARSEC benchmark suite [2] with its `simlarge` input for our C/C++ implementation, and for Java we used 6 applications from the DaCapo benchmark suite [3], with default inputs, and all the Java Grande benchmarks [28], with size A inputs. We ran PARSEC and Java Grande with 8 threads; we let the DaCapo benchmarks self-configure based on the number of processors and did not instrument the DaCapo harness. We also ran 4 additional full applications, each with 8 threads: `mysql`, a database server, tested using the sysbench OLTP benchmark with the default table size (10,000) for the performance measurements and table size 100 for debugging; `apache`, a web server, tested using ApacheBench; `aget`, a download accelerator, tested fetching a large web file; and `pbzip2`, a compression tool, tested compressing a 100MB text file. For performance measurements, we ran the uninstrumented version and Recon, with the *first-read*, *first-write*, and *racy-lookups* optimizations. We also ran three less-optimized configurations to understand the impact of each optimization: "Base" analyzes all memory accesses; "FR" uses just the *first-read* optimization; "FR/W" adds the *first-write* optimization. We ran all experiments on an 8-core 2.8GHz Intel Xeon with 16GB of memory and Linux 2.6.24. The Java tool used the OpenJDK 64-bit Server VM 1.6.0 with a 16GB max heap. We report results averaged over 10 runs of each experiment.

6.2 How Effectively Does Recon Find Bugs?

We produced reconstructions using graphs collected from 25 buggy runs and 25 non-buggy runs. We ranked the reconstructions as described in Section 4.2. We examined the highest-ranked reconstruction that illustrated the bug and analyzed the key properties of that reconstruction. Table 3 summarizes our findings.

False Positives. The most important result in Table 3 is that for all applications, *the top-ranked reconstruction revealed the bug*, as shown in Column 2. This result demonstrates that our ranking technique effectively directs programmer attention to buggy code with no distracting false positives. This result also corroborates the results from Section 4.2, showing that our features precisely isolate buggy reconstructions.

Unrelated Code in Reconstructions. Columns 3 and 4 in Table 3 show the number of relevant and irrelevant code points that were included in the bug's reconstruction. We consider a code point related if it performs a memory access that reads or writes a corrupt or inconsistent value, or if it is control- or data-dependent on the buggy code. In most cases, virtually all code in the reconstruction is relevant to the bug. However, some reconstructions include code points unrelated to their bug. In `aget`, the two irrelevant code points are in straight-line code sequences with related code points, at a distance of less than five lines. Such nearby but irrelevant code is not likely to confuse a programmer.

In `mysql`'s case, five out of seven unrelated code points are in straight-line sequences with relevant code. The remaining two in `mysql`'s reconstruction, and all five in `apache`'s reconstruction, were not in straight-line code with relevant points. Instead, they were in another function that was the caller or a callee of a function containing relevant code. Developers debugging programs are likely to understand such caller-callee relationships, suggesting that these code points will not be too problematic.

`weblech` had several irrelevant code points in its reconstruction (28). The reason for their inclusion is that the bug usually occurs at the start of the execution. At this point, constructors have only just initialized data at a variety of code points in the program, resulting in many edges being added between initialization code and other code. The initialization code is easy to identify, especially with program knowledge. These code points clutter the reconstruction, but the bug is reported accurately.

Program	Rank of Bug	# Code Pts. Rel.	Irr.	In Order?	Code Pts Missing	Sensitivity To # Buggy w/ 5	w/ 15	Collect Time (h:m:s)
logandswp	1	6	1	Yes	0	1	1	—
circlist	1	3	3	Yes	0	1	1	—
textreflow	1	8	0	Yes	0	1	1	—
jsstrlen	1	7	0	Yes	0	1	1	—
apache	1	5	5	Yes	0	1	1	0:27:32
mysql	1	8	7	Yes	0	34	9	0:07:08
pbzip2	1	11	0	Yes	1	2	1	1:51:56
aget	1	4	2	Yes	0	8	1	0:59:41
stringbuffer	1	6	0	Yes	0	1	1	—
vector	1	6	0	Yes	0	1	1	—
weblech	1	6	28	Yes	0	4	1	0:13:36

Table 3. Properties of reconstructions. Column 1 is the rank of the bug's reconstruction. Columns 2 and 3 show the number of relevant and irrelevant code points in the bug's reconstruction. Column 4 shows whether the bug's reconstruction was in order. Column 5 shows the number of relevant code points missing from the reconstruction. Columns 6 and 7 show the rank of the bug's reconstructions using only 5 and 15 buggy graphs. Column 8 shows the graph collection time.

Reconstruction Order Accuracy. Column 5 shows whether or not the code points in the reconstruction were shown in the order leading to buggy behavior. Code points appear in an order that leads to buggy behavior in all cases. In `logandswp`, the last code point in the buggy execution order appears in both the prefix and suffix of the reconstruction, because the code point is in a loop, however, the buggy interleaving is clear.

Missing Code Points. Column 6 shows the number of code points directly involved in the bug that were omitted from the reconstruction. Only one case lacked any involved code points: the code points in `pbzip2`'s reconstruction all relate to establishing the corrupted state condition required for a crash to occur. The actual crashing access is not included.

Sensitivity to Number of Buggy Runs. Columns 7 and 8 illustrate Recon's sensitivity to the number of buggy runs used. Column 7 shows the rank of the bug's reconstruction using 25 nonbuggy runs and 5 buggy runs. Column 8 shows the rank using 25 nonbuggy and 15 buggy runs. Even with very few buggy runs, Recon gives a high rank to reconstructions of the bug. Using fewer buggy runs does not impact precision substantially, except for `mysql`. Excluding `mysql`, Recon ranked the bug's reconstruction 8th or better with just 5 buggy runs, and first with 15 buggy runs. For `mysql`, using fewer runs caused Recon to rank some nonbuggy reconstructions above the bug's — 33 with 5 buggy runs, and 8 with 15 buggy runs. As shown in Column 2, Recon *always* ranked the bug's reconstruction first with 25 buggy runs. These results show that with very few buggy runs, Recon can find bugs with high precision. In cases where a small number of buggy runs is insufficient, adding more runs increases Recon's precision.

Graph Collection Time. Column 9 shows that the total time required to collect 25 buggy and 25 nonbuggy graphs is not prohibitively long. In our experiments, all applications took under two hours; `apache`, `mysql`, and `weblech` all took under 30 minutes. These data show that Recon is not only effective at detecting real bugs in these full applications, but also reasonably fast. In Section 6.4 we characterize the overheads our technique imposes over uninstrumented execution.

6.3 Case Study: Debugging an Unresolved Bug

The `weblech` bug is open and unresolved in the program's bug repository. While the bug has been discussed previously [12], we were unaware of any details of the bug prior to this case study. We used Recon to find the problem, and we were able to write a fix using Recon's output and our limited program knowledge.

We began with a bug report describing intermittent non-termination. Using the input from the report, we were able to reproduce the bug in about 1 in 15 runs. We then ran the application repeatedly and watched the output to identify the hang. We noticed that, consistently, at least one thread crashed on a null pointer dereference during hanging runs. We collected 25 buggy and 25 nonbuggy runs, identifying bugginess by watching for unhandled exceptions. We then produced reconstructions from these runs.

The first reconstruction reported was related mostly to object constructors, but also included evidence of several accesses to a shared queue data structure, as well as a suspicious `while` loop termination condition involving the queue's size. The body of the reconstruction contained the initialization of and accesses to the size of the queue. The sink of the reconstruction's edge was an access to the queue data structure in the dequeue method. In the suffix of the reconstruction was another call to the queue's dequeue method. As we described in Section 2.1, such an interleaving of a dequeue call between an access to the queue's size and a subsequent dequeue call violates the atomicity of the pair of operations. The atomicity violation leads to a thread crashing early due to the `NullPointerException` we observed. Crashing prevents the thread from correctly updating the variable for the `while` loop to read. The crash is therefore also responsible for the program's non-termination, as described in the bug report. We fixed the bug by extending a `synchronized` block including the queue size check and the dequeue. With our fix, we didn't see the buggy behavior in several hundred runs — we conclude that we fixed the bug based on the information provided by Recon.

6.4 Performance

In Table 4, we report runtimes relative to uninstrumented execution for Recon and the three less-optimized configurations. In the best case, Recon imposes slowdowns as low as 34% for C/C++ (`pbzip2`) and 13% for Java (`weblech`).

Slowdown for full applications never exceeds 24x, even during an industrial strength test of a commercial database (`mysql`). For PARSEC, we saw slowdowns ranging from 5.5x to 28x, showing that Recon performs well on applications with a variety of sharing patterns. We saw comparable results for DaCapo: overheads of Recon ranged from 5.6x to 17.3x.

Interestingly, overheads tended to be more severe for applications that perform *infrequent sharing*, than those that share often.

	Name	Recon	FR/W	FR	Base		Name	Recon	FR/W	FR	Base
			Slowdown (x)						Slowdown (x)		
Apps.	weblech	1.1	1.2	1.2	1.1	PARSEC	dedup	5.5	5.8	5.8	13.8
	pbzip2	1.3	1.3	1.3	1.5		canneal	6.8	6.8	6.5	14.9
	aget	1.9	1.9	1.9	1.9		freqmine	8.8	52.6	56.6	223.8
	apache	5.4	31.7	31.7	177.1		fluidanimate	9.8	9.9	10.1	9.8
	mysql	23.9	102.1	127.2	129.9		streamcluster	10.1	10.1	10.3	10.1
DaCapo	pmd	5.6	5.8	6.0	6.3		blackscholes	14.4	17.8	18.0	40.9
	avrora	5.6	7.9	10.3	27.8		ferret	14.6	70.9	73.1	537.3
	tomcat	6.9	6.2	6.7	9.1		bodytrack	14.9	116.2	120.8	595.5
	xalan	7.1	7.0	7.5	10.8		facesim	15.8	18.8	19.2	29.2
	luindex	8.2	9.1	14.3	20.6		swaption	17.9	96.0	100.6	383.7
	lusearch	17.3	18.1	22.7	22.6		x264	18.9	218.4	236.8	697.4
Java Grande		74.9	85.1	88.4	563.7		vips	28.8	230.6	257.6	996.8

Table 4. Performance of Recon and less-optimized configurations relative to uninstrumented execution.

For example, `dedup`, which uses shared queues, and `avrora`, which exhibits a high-degree of fine-grained sharing [3], both had fairly low overheads, around 6x. In contrast, `swaptions` has infrequent synchronization [2] and threads in `lusearch` interact very little [3] — both suffered higher overheads, around 18x. This trend is further illuminated by the Java Grande benchmarks; these are primarily data-parallel scientific computations that perform little sharing [28]; their average overhead is 75x. Nonetheless, Recon is efficient in applications with high-frequency sharing and for all the mainstream applications we tested.

Effectiveness of Optimizations. Comparing "FR" with "Base" and "Recon" with "FR/W" in Table 4, we see that the first-read and racy-lookup optimizations, respectively, significantly improve performance. Comparing "FR/W" and "FR", we see that the first-write optimization has less significant effect in general — likely because writes are less common than reads — but for `mysql` and `lusearch`, the first-write optimization is clearly important.

The data show that our optimizations are essential to Recon's efficiency. For many applications, our optimizations reduce Recon's slowdown by orders of magnitude. `apache` is one such application: without optimizations, `apache`'s slowdown is 177x, making full-scale tests nearly impossible due to timeouts and unhandled delay conditions in the code. Optimizations reduce this to just 5.4x, enabling Recon to be used with real bug-triggering inputs.

In our experiments, we used PARSEC's `simlarge` inputs to make experimenting with unoptimized configurations feasible, but there is no need to scale inputs for use with Recon. We also experimented with PARSEC's `native` input, using Recon with all optimizations. Experiments finished quickly, and we saw slowdowns nearly identical to the `simlarge` input.

The optimizations have less impact on our Java implementation, but still account for significant speedups (*e.g.*, `avrora`, `luindex`). For most Java benchmarks, the racy-lookup optimization had little effect. Java uses several techniques that significantly reduce the cost of acquiring locks [13]. It is likely that the racy-lookup optimization is less beneficial than in C/C++ because the cost of locking is lower in Java to begin with.

Memory Overhead. The C/C++ Recon implementation uses a fixed-size 4GB metadata table, dominating memory overheads in our experiments. Graphs are small in comparison. The table is large enough that the impact of hash collisions was negligible. In a memory-constrained setting, a smaller table could be used at the expense of decreased precision due to hash collisions. In Java, each field and array element is shadowed by a metadata location: memory overhead scales roughly linearly with the program's footprint. Peak overheads in the optimized version ranged from 2.5x to 16x.

7. Related Work

Prior work has explored a variety of atomicity violation detection approaches. AVIO [17] is an invariant-based approach that infers from a set of training runs which unserializable interleavings are allowed in correct executions. In subsequent runs, AVIO reports any unserializable interleavings not present in the invariant set as possible atomicity violations. AVIO focuses on single-variable atomicity violations. SVD [30] attempts to infer atomic sections based on data and control dependences and determines whether an execution is serializable with respect to those sections. Velodrome [8] is a sound, precise dynamic atomicity checker that reports an error if an execution of a program with explicit atomic blocks is not conflict-serializable.

Recent work introduced general approaches to detecting concurrency errors. Bugaboo [18] first proposed the use of context-aware communication graphs for debugging general concurrency errors. DefUse [27] employs a similar communication-based strategy, by finding communication invariant violations related to errors. Neither approach provides programmers with information about the actual interleaving schedule at the root cause of the bug. Context-aware graphs, and DefUse invariants only isolate a single communication event related to a bug. One of Recon's main contributions is that it provides information about the execution schedule encoded in a reconstruction. Additionally, Bugaboo and DefUse rank anomalies along a single axis of suspiciousness. In contrast, Recon ranks reconstructions along several axes, using the features described in Section 4. Finally, Recon's performance is orders of magnitude better than the performance of Bugaboo's software implementation. DefUse does not show overheads for a standard benchmark suite, so direct comparison is impossible, but Recon's performance on reactive applications is comparable.

Interleaving Constrained MultiProcessor [32] (PSets) proposes architecture support for dynamic bug avoidance. PSets uses test runs to determine happens-before invariants on memory operations and tries to enforce them by manipulating the schedule. This technique handles several types of single-variable bugs. Falcon [23], uses a library of bug patterns to identify potential concurrency bugs; it watches a stream of memory operations to single addresses and searches for a matching pattern. Falcon's technique is mostly applicable to single-variable concurrency bugs. Hammer, *et al.* propose pattern-based dynamic analysis for detecting bugs related to atomicity properties of accesses to sets of variables [10].

Recon is set apart from prior work by the fact that reconstructions are general, handling a variety of single- and multi-variable errors without relying on characteristics of specific bug types. Furthermore, reconstructions illustrate bugs more clearly, showing a focused portion of the interleaving schedule near the bug's cause.

Inter-thread communication invariants have also been exploited to specify correct communication behavior at the function or method level and check that a program execution conforms to this specification [29]. In contrast, Recon focuses on communication at the instruction granularity and helps programmers determine the cause of unexpected behavior rather than helping them enforce expected invariants.

Another way of dealing with concurrency bugs is explorative testing to expose buggy executions. CHESS [21] explores executions by interposing on synchronization operations; its goal is to expose buggy executions during testing. Burckhardt *et al.* propose a scheduling technique that probabilistically exposes bugs [5]. CTrigger [22] is a heuristic to expose atomicity bugs. The goal of these techniques is not to detect or explain bugs but to expose buggy executions. In contrast, Recon helps programmers understand bugs by reconstructing the interleavings that likely led to bugs' symptoms. Explorative testing is complementary to our approach.

Liblit *et al.*'s Cooperative Bug Isolation (CBI) [15] uses sampling techniques to collect information from deployed applications. Based on sets of labeled passing or failing runs, CBI finds code points related to failures. CCI [11] extends these sampling techniques to concurrency bug patterns. In contrast, Recon does not require deployment-scale data to efficiently detect bugs. It provides more information about buggy interleavings, and doesn't rely on bug-specific patterns.

8. Conclusion

In this paper we introduced Recon, a novel and *general* approach to isolating and understanding all types of concurrency bugs. Recon works by reconstructing fragments of buggy executions that are likely the result of a bug, providing sufficient yet succint information to help programmers understand the cause of concurrency bugs, rather than just showing the code involved or reproducing an entire buggy execution.

Reconstructions show the schedule of execution that led to the bug, clearly exposing its root cause. Reconstructions are built by observing multiple executions of a program and collecting timestamped communication graphs that encode information about the ordering of inter-thread communication events. We developed a simple machine-learning approach to identify buggy reconstructions. We proposed three bug-independent features of reconstructions that together precisely isolate reconstructions of buggy executions. In order to provide efficient collection of timestamped graphs, we used several techniques that significantly reduce runtime overheads. We implemented Recon for C/C++ and Java and evaluated it using large software. Our results show Recon reconstructs buggy executions with virtually no false positives, and that collecting the data comprising reconstructions takes just minutes.

Acknowledgments

We thank the anonymous reviewers and the Sampa group for their helpful feedback. Thanks to Joseph Devietti, Laura Effinger-Dean, Colin Gordon, Dan Grossman, and Karin Strauss for their invaluable feedback on the manuscript. Special thanks go to Zachary Rait and Julian Knutsen for early work on the Recon infrastructure. This work was supported in part by an IBM PhD fellowship, an ARCS Foundation Fellowship, a Microsoft Research Faculty Fellowship, NSF grant CCF-0811405, and gifts from Intel.

References

[1] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *PLDI*, 2008.

[2] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Princeton University, January 2008.

[3] S. M. Blackburn et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, 2006.

[4] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

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

[6] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[7] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, 2010.

[8] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, 2008.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 2009.

[10] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE*, 2008.

[11] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[12] P. Joshi and K. Sen. Predictive Typestate Checking of Multithreaded Java Programs. In *ASE*, 2008.

[13] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, 2002.

[14] I. Kononenko. Estimating Attributes: Analysis and Extensions of RELIEF. In *European Conference on Machine Learning*, 1994.

[15] B. Liblit. *Cooperative Bug Isolation*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

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

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

[18] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-Aware Communication Graphs. In *MICRO*, 2009.

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

[20] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.

[21] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

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

[23] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault Localization in Concurrent Programs. In *ICSE*, 2010.

[24] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.

[25] M. Ronsee and K. D. Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ToCS*, 1999.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ToCS*, 1997.

[27] 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.

[28] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Supercomputing*, 2001.

[29] B. P. Wood, A. Sampson, L. Ceze, and D. Grossman. Composable Specifications for Structured Shared-Memory Communication. In *OOPSLA*, 2010.

[30] M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *PLDI*, June 2005.

[31] M. Xu, M. D. Hill, and R. Bodik. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *ASPLOS*, 2006.

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

[33] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.