Finding Concurrency Bugs with Context-Aware Communication Graphs

Brandon Lucia Luis Ceze

Department of Computer Science and Engineering, University of Washington

{blucia0a,luisceze}@cs.washington.edu
http://sampa.cs.washington.edu

ABSTRACT

Incorrect thread synchronization often leads to concurrency bugs that manifest nondeterministically and are difficult to detect and fix. Past work on detecting concurrency bugs has addressed the general problem in an ad-hoc fashion, focusing mostly on data races and atomicity violations.

Using graphs to represent a multithreaded program execution is very natural, nodes represent static instructions and edges represent communication via shared memory. In this paper we make the fundamental observation that such basic context-oblivious graphs do not encode enough information to enable accurate bug detection. We propose *context-aware communication graphs*, a new kind of communication graph that encodes global ordering information by embedding communication contexts. We then build *Bugaboo*, a simple and generic framework that accurately detects complex concurrency bugs. Our framework collects communication graphs from multiple executions and uses invariant-based techniques to detect anomalies in the graphs.

We built two versions of Bugaboo: BB-SW, which is fully implemented in software but suffers from significant slowdowns; and BB-HW, which relies on custom architecture support but has negligible performance degradation. BB-HW requires modest extensions to a commodity multicore processor and can be used in deployment settings. We evaluate both versions using applications such as MySQL, Apache, PARSEC, and several others. Our results show that Bugaboo identifies a wide variety of concurrency bugs, including challenging multivariable bugs, with few (often zero) unnecessary code inspections.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors);
D.2.5 [**Software Engineering**]: Testing and Debugging;
D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Design, Reliability

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.
MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

1. INTRODUCTION

As multithreaded software becomes ever more important, we need to enable programmers to productively write and debug concurrent programs. This has motivated the development of several methods and tools for debugging concurrency errors.

There are several categories of concurrency bugs. The main categories discussed in the literature are data races, locking discipline violations, atomicity violations and ordering violations. Past work on concurrency error debugging typically covered each category separately. For example, RecPlay [17] detects data races and provides replay capabilities; AVIO [9] and Atom-Aid [10] detect atomicity violations using heuristics based on identifying unserializable interleavings; and Eraser [18] detects locking discipline violations using its lock-set algorithm. While these systems greatly help in identifying concurrency bugs, they address the general problem in a piecemeal way. Moreover, current tools do not adequately address less well-studied classes of bugs such as ordering violations and bugs involving multiple variables [8].

Communication graphs are a convenient representation of a multithreaded program execution. In a basic communication graph, nodes represent memory instructions and edges represent communication via shared-memory. Concurrency errors lead to abnormal inter-thread communication and may therefore manifest themselves as anomalies in communication graphs. Because multithreaded execution is nondeterministic and consequently bugs manifest intermittently, different executions lead to different graphs. By examining the differences between many graphs, it is possible to identify anomalous communication and consequently where bugs are likely to be. The biggest advantage of this approach is that it is *general*, as it does not rely on heuristics that are specific to a class of bugs. The key challenge, however, is building a communication graph in which enough relevant information is encoded. If insufficient information is encoded, bugs may not render as graph anomalies.

In this paper, we make the fundamental observation that basic communication graphs do not encode enough information to enable general concurrency bug detection. We address this problem by proposing *context-aware communication graphs*, a new kind of communication graph that uses communication context to encode access ordering information. Communication contexts are formed by capturing the sequence of all recent communication events observed by a thread. We then develop *Bugaboo*, a complete system that leverages these graphs to provide efficient and accurate bug detection, useful both in development and deployment situations.

This paper makes the following contributions: (1) we propose context-aware communication graphs; (2) we propose two invariant-based approaches to processing communication graphs and accurately locating bugs in code: one fully automatic and one semi-

```
int  length;  // protected by lock L
char *str;    // protected by lock L

Thread 1              Thread 2
...                   ...
lock(L);              lock(L);
tptr = str;           str = newstr;
unlock (L);           unlock(L);
...                   ...
lock(L);              lock(L);
tlen = length;        length = 15;
unlock(L);            unlock(L);
```

(a)
Multivariable atomicity violation.
Thread 1 reads inconsistent **str/length**.

```
Thread 1                                  Thread 2
void qDelete (Q *q)                       void *decompress_consumer(. . .)
{                                         {
 done = true;                              for(;;)
 ...                                       {
 pthread_cond_destroy(q->notEmpty);         ...
 delete q->notEmpty;                        pret = pthread_cond_timedwait(q->notEmpty,
 ...                                        ...                             q->mut,
}                                                                           &waitTimer);
                                            if(done) return 0;
                                           }
                                          }
```

(b)
Ordering violation. If **qDelete()** in Thread 1 is called before Thread 2 is waiting on the conditional variable, program crashes.

Figure 1: Examples of a multivariable atomicity violation and an ordering violation. Dashed arrows represent buggy interleaving.

automatic; (3) we describe *BB-SW*, a software-only implementation of Bugaboo and propose *BB-HW*, which is based on a set of architecture extensions to a commodity multicore system that brings performance overheads in collecting context-aware communication graphs to nearly zero; (4) we show how BB-HW can be used in production runs; and (5) we evaluate BB-SW and BB-HW and show that they are able to identify complex bugs (including multivariable ones), requiring just a few code inspections by the programmer.

The remainder of this paper is organized as follows. Section 2 describes common types of concurrency errors. Section 3 shows why context-oblivious communication graphs limit generic bug detection and explains how our context-aware communication graphs solve this problem. Section 4 describes Bugaboo's implementation. Section 5 elaborates on our invariant-based debugging framework. Section 6 evaluates bug detection accuracy and characterizes both of our implementations. Finally, Section 7 discusses related work and Section 8 concludes.

2. CONCURRENCY BUGS

There are several types of concurrency errors. Arguably, the most well known is a data race. A data race occurs when two or more memory operations in different threads, at least one of which is a write, access the same memory location and are not ordered by a happens-before relationship (synchronization). Absence of data races, however, does not imply lack of concurrency defects. The other major types of non-deadlock concurrency bugs are atomicity violations and ordering violations, and both can occur in the absence of data races.

Atomicity violations [4] happen when memory operations supposed to be executed atomically are not enclosed inside the same critical section. Figure 1(a) shows an atomicity violation involving multiple variables: the update of `str` and `len` in Thread 2 should be atomic but they interleave the read accesses in Thread 1, which will get inconsistent data — old value of `str`, but new value of `len`.

Ordering violations happen when memory accesses in different threads happen in an unexpected order. In Figure 1(b) we show an example: if `qDelete()` in Thread 1 is called immediately before Thread 2 waits on the conditional variable (call to `pthread_cond_timedwait()`), the program crashes. In a correct execution, the call to `pthread_cond_timedwait()` in the final iteration (`done = true`) of the `for` loop in Thread 2 should be ordered before the call to `qDelete()` in Thread 1, but this constraint is absent in the code.

Lu et. al [8] did a comprehensive study of concurrency bugs that appear in open-source applications such as Mozilla, MySQL and Apache. The study showed that the vast majority (97%) of non-deadlock concurrency bugs are either atomicity violations or ordering violations. Moreover, the study points out that one-third of the non-deadlock concurrency bugs involved multiple variables. Most previous work on concurrency bug detection focused on data races [15, 17], locking discipline violation [18] and atomicity violations [9, 10, 19]. General multivariable bugs and ordering violations have not been thoroughly addressed in prior work. We aim at providing a completely general framework for concurrency bug detection by not relying on any heuristics specific to one type of bug.

3. CONTEXT-AWARE COMMUNICATION GRAPHS

3.1 Overview

The approach to concurrency bug detection we are exploring involves collecting inter-thread communication graphs from multiple executions, and then processing them to detect anomalies that are likely the result of concurrency bugs. There are many ways to process graphs, all of which depend on the fact that the presence or absence of certain edges distinguish a correct execution's graph from an incorrect execution's graph. This way, graph differences directly reflect buggy communication.

Figure 2 illustrates this concept. Figure 2(a) shows an ordering bug taken from MySQL-4.1.8. Figure 2(b) shows the communication graphs obtained from a correct (top) and an incorrect execution (bottom). In the incorrect execution there is no communication between the store to `dynamicId` in Thread 1 and the load in Thread 2, so Thread 2 reads uninitialized data. Comparing these graphs directly points to the communication anomaly in the execution. We discuss graph processing in more detail in Section 5.

The key in using communication graphs for bug detection is how to build the graph and consequently what information is encoded. We now show that a basic context-oblivious communication graph does not encode sufficient information for a general approach to bug detection. We then show how the new graph abstraction we propose addresses this problem.

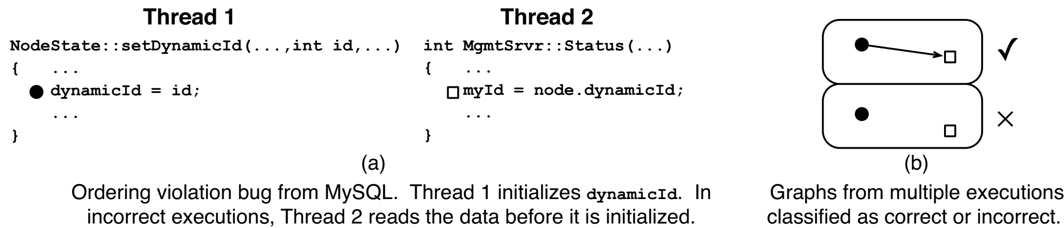


Figure 2: High-level view of detecting concurrency bugs based on communication graph anomalies. Markers represent memory operations involving shared data.

3.2 Context-Oblivious Communication is Insufficient

In a basic, context-oblivious, communication graph, concurrency bugs may lead to edges that are only present in graphs of buggy executions, therefore simple graph differences might point to bugs. For example, consider the bug in Figure 3(a), extracted from MySQL: if the read access of `log_type` in Thread 2 interleaves between the two writes in Thread 1, the bug manifests itself. Figure 3(b) shows a communication graph resulting from a union of graphs from a correct and an incorrect execution. The bad interleaving in (a) leads to an edge (dashed) in (b) that only appears in buggy executions.

Even though context-oblivious graphs can be used to detect some bugs, they can not be used to detect many other bugs. This is because many buggy interleavings lead to edges that are also present in graphs from correct executions. Figure 4(a) shows a typical multivariable concurrency bug, in which a string/length pair of variables is not updated atomically. The read accesses of `str` and `len` in Thread 2 will get inconsistent data if they interleave the write accesses in Thread 1. However, as Figure 4(b) shows, this interleaving does not lead to a unique edge in the communication graph, making it impossible to detect the bug by considering graph differences. In summary, the edges in the communication graph that are the result of the bad interleaving are also present in a graph from a correct execution. What is missing in this basic context-oblivious communication graph is a notion of *relative order* of communication events — nodes solely represent static instructions, and there is no notion of order between edges.

3.3 Embedding Ordering in Communication Graphs

One direct way of guaranteeing that there is enough ordering information in a communication graph is to have each node represent a *dynamic* memory operation. This is clearly impractical because the graph size would be unbounded, growing with execution time. We propose a new kind of communication graph that encodes information about the relative order of communication but is bounded in size and not significantly larger than a basic, context-oblivious, communication graph.

The key aspect of the graph we propose is that nodes represent a combination of static memory instruction and the communication context when the instruction was executed. We call these context-aware communication graphs. The communication context of a memory instruction is the sequence of communication events observed by a thread immediately prior to the execution of the memory instruction; it is obtained by monitoring the communication events observed by a thread, regardless of the data address involved, and inserting them in a fixed-size FIFO queue.

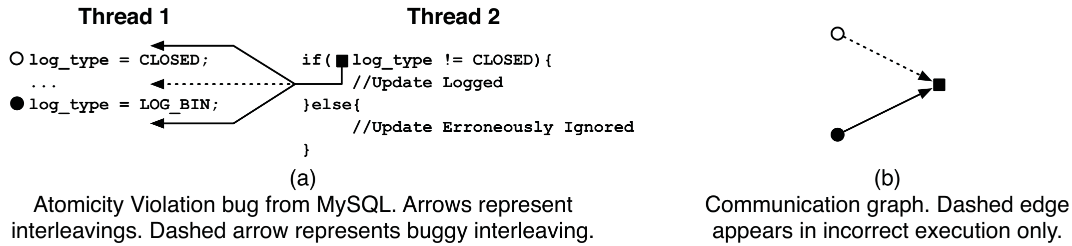
There are four types of communication events observed by a local thread: (1) *LcRd*, a read of data recently written by a remote thread; (2) *LcWr*, a write to data recently read by a remote thread; (3) *RmRd*, a remote read of data recently written locally; and (4) *RmWr*, a remote write to data recently read or written locally. The contents of the FIFO queue is the context. Note that the communication events that make up the context are not the result of accesses to a specific data address. This is exactly what we need because we want to capture the notion of global ordering across memory accesses irrespective of data address. Context size is arbitrary, and the longer it is, the more ordering information is encoded in the graph.

Formally, a context-aware communication graph is defined as $G = (V, E)$, where $v \in V$ is a tuple $(inst, ctx)$, and each edge $(u, v) \in E$ is a pair of these tuples.

An edge $((u.inst, u.ctx), (v.inst, v.ctx))$ is present in G if during the execution from which G was constructed, $u.inst$ communicated with $v.inst$, and when $u.inst$ executed its processor context was $u.ctx$ and when $v.inst$ executed its processor context was $v.ctx$.

Each node in a context-oblivious communication graph maps to multiple nodes in a context-aware communication graph from the same execution. More precisely, if the context-oblivious communication graph of an execution has N_s nodes, a context-aware communication graph of the same program execution will have at most $N_s \times C$ nodes, where C is number of all possible contexts in which a memory access instruction can execute. Since there are four types of events, $C = 4^S$, where S is the context size. We experimentally determined (Section 6.2.1) that a context of five events (1024 possible contexts) is enough to capture enough ordering to detect all types of concurrency bugs discussed in the literature. In practice, the addition of context does not mean that a context-aware communication graph is 1024 times bigger, since each node executes in a small fraction of the possible contexts. Our experiments (Section 6.5) never showed an increase larger than 50-fold.

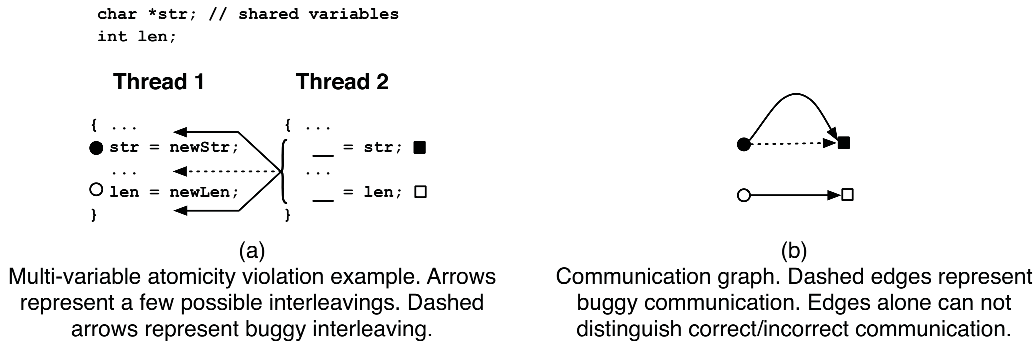
Figure 5 shows how context-aware communication graphs can be used to reveal concurrency bugs. Figure 5(a) shows multiple executions of the buggy code in Figure 4(a). For each execution, it shows the memory access interleavings and the symbols in parenthesis represent the communication context of each thread at the point when the communication happened. For example, refer to the first execution (top) in Figure 5(a): the first operation on the left (write to `str`) was executed after two remote reads had been received by the local node, so the context at that point is (RmRd, RmRd); after that, the context becomes (RmRd,RmRd,LcWr) at the point when the next operation on the left is executed (write to `len`). Note how the context adds ordering information to the communication graph: Figure 5(b) shows that it is now possible to distinguish in the graph communication between write to `str` and read from `str` depending on when in the execution the communication happened, unlike the basic communication graph in Figure 4(b). Recall that basic



(a) Atomicity Violation bug from MySQL. Arrows represent interleavings. Dashed arrow represents buggy interleaving.

(b) Communication graph. Dashed edge appears in incorrect execution only.

Figure 3: Atomicity violation example and how a communication graph can reveal the bug.



(a) Multi-variable atomicity violation example. Arrows represent a few possible interleavings. Dashed arrows represent buggy interleaving.

(b) Communication graph. Dashed edges represent buggy communication. Edges alone can not distinguish correct/incorrect communication.

Figure 4: A basic communication graph is often not enough to detect concurrency bugs. Edges can not distinguish correct and incorrect executions.

communication graphs are limited in exposing concurrency bugs because there are no edges that are only present in the graphs of incorrect executions. Conversely, with context-aware graphs there are edges that are only present in graphs of incorrect executions.

4. IMPLEMENTATION

4.1 BB-SW: Software-Only Bugaboo

BB-SW uses binary instrumentation to monitor memory accesses and build the communication graph. It has three key data-structures: (1) a table that maps each memory location to the instruction address, thread ID and communication context of the last writer, which can be configured to use word or line granularity memory addresses; (2) an array that keeps the communication context of each thread; and (3) the context aware communication graph itself.

Whenever a thread reads from or writes to a memory location whose last writer was not itself, BB-SW adds a new edge to the graph. The edge's source is the last writer (instruction address, context) and the edge's sink is the current instruction plus its thread's current context. This policy captures both WAW and RAW edges.

To maintain the communication context, whenever a thread accesses a location last written by another thread, it records a corresponding LcRd or LcWr event identifier in its context FIFO queue. The thread which last wrote the memory location records a corresponding RmRd or RmWr event in its context. The size of the event FIFO queue is fixed at five and when full, the oldest element is discarded.

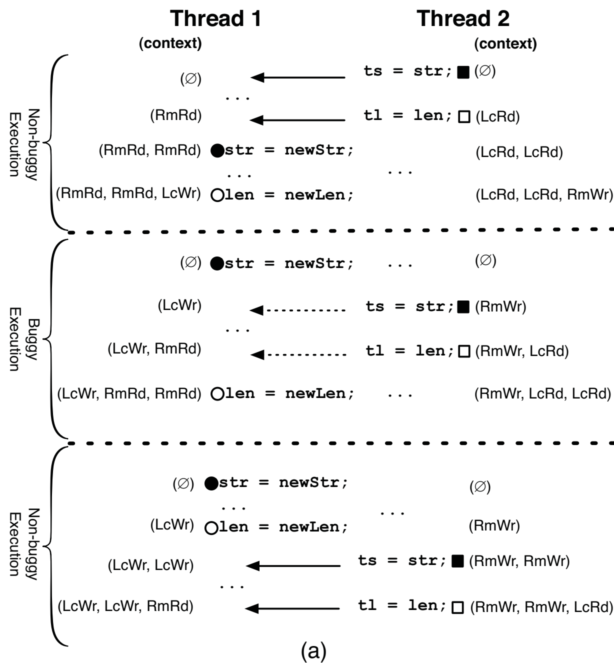
4.2 BB-HW: Architectural Support for Context-Aware Communication Tracking

While BB-SW provides acceptable performance for debugging, collecting communication graphs purely in software can lead to significant performance degradation. We now describe BB-HW, which uses a set of modest hardware extensions to reduce overheads in monitoring communication between memory instructions and to keep track of context. Actual graph processing for debugging in BB-HW still happens in software.

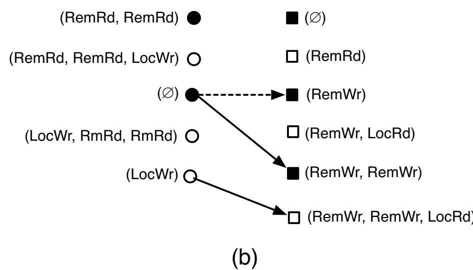
The cache coherence protocol in typical shared-memory multiprocessor systems already provides most of the support BB-HW needs, since all communication carried between processors happens via coherence messages. Broadly, beyond what a typical MESI coherence protocol offers, we need information about the instruction addresses that lead to coherence messages and a way to store communication events as they happen, such that a software component can periodically read them. BB-HW has five components: (1) a per-processor context register that keeps track of recent communication events; (2) coherence message extensions to carry producer/consumer instruction information; (3) cache-line extensions (meta-data) to keep track of producer/consumer instructions and context; (4) a software-visible table to store communication edges as they happen; and (5) a thin software component that periodically reads the graph edges collected. Figure 6 shows an overview of BB-HW's extensions to a commodity multiprocessor.

4.2.1 Keeping Track of Context

The four communication events discussed in Section 3.3 map directly to cache coherence events, since we only consider cache-to-cache transfers as communication. We give each relevant cache coherence event a two-bit code: local read miss (LcRd); local write miss or upgrade miss (LcWr); incoming invalidate request (RmWr); and incoming read request (RmRd). Recall that to maintain the communication context we keep a FIFO queue of recent communication events. The context register itself is a simple shift register, and the event code is shifted in as the event happens. We want to encode global communication across addresses, therefore we dis-



A few of the possible interleavings and their corresponding communication contexts from code in Figure 4(a). Dashed arrows correspond to bad interleavings.



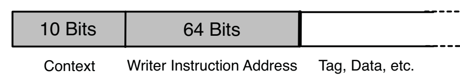
Context-aware communication graph. Dashed edges come from bad interleavings.

Figure 5: Context-aware communication graph example revealing the multivariable atomicity violation in Figure 4(a), which can *not* be revealed by a basic context-oblivious communication graph.

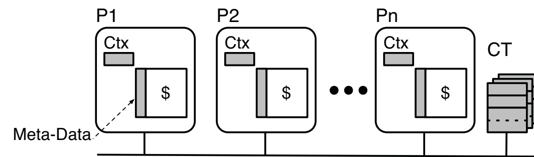
regard the data address of the events. Each processor in the system has its own context register (Figure 6(b)). As in BB-SW, we keep five events of context, so our context registers are 10 bits long.

4.2.2 Cache Meta-Data and Coherence Extensions

Precisely keeping track of communicating instructions requires keeping last-writer information at the granularity of words (or potentially bytes) and for the whole memory. This is clearly too expensive to do in hardware. We chose two simplifications to reduce hardware complexity at the cost of some information loss. First, we track communication at cache-line granularity. False sharing might lead to edges that are not actual communication. Second, we only monitor inter-thread communication that happens via cache-to-cache transfers, which might miss edges that are present in the actual communication. As our data shows (Section 6.3), the information loss is not significant and does not limit the bug detection capability of our techniques.



(a)
Extensions to a typical cache line.



(b)
Extensions to a bus-based commodity multiprocessor.

Figure 6: BB-HW architectural extensions (shaded) to a typical multiprocessor.

We add meta-data to each cache line to keep track of the instruction address (virtual address) and context when the line was written to, which is the source of a communication edge. The meta-data includes a field that stores the instruction address (i.e., 64 bits) and a field that stores the context (10 bits) from when the data was produced, i.e., when the store instruction was executed. Therefore, the total overhead per cache line is $64 + 10 = 74$ bits per cache line[1]. Figure 6(a) illustrates our extensions to the cache line.

The meta-data is updated when a processor writes to a line that is not in exclusive nor modified state, i.e., during a write or upgrade miss. As a result, meta-data updates are only as frequent as write/upgrade misses. Without loss of generality, we assume the underlying system has a MESI cache coherence protocol. We augment the following coherence messages to include information about the instructions involved in the communication:

Read reply. The supplier sends the meta-data of the corresponding line. This keeps track of read-after-write (RAW) communication.

Invalidate reply (ack). The supplier sends the meta-data of the line that was invalidated. This is used to keep track of write-after-write (WAW) communication.

4.2.3 Communication Table

The purpose of the *Communication Table (CT)* is to temporarily store communication edges as they happen before the software component reads them. It is organized as a simple queue, where each entry contains the instruction address and context of both the *source* and *destination* of a communication edge. Thus, the size of each entry is $(64 + 10) \times 2 = 148$ bits. The CT, shown in Figure 6(b), can be organized either as a centralized or distributed data-structure. Since there are no global consistency properties that need to be kept, each processor can have its own CT, posing no scalability issues.

Three events leads to a write to the CT:

Read reply. The source is set to the meta-data of the supplied line. The destination is set to the instruction address of the local load instruction that caused the miss and the context when the miss happened. This captures RAW edges.

[1]The instruction address can be easily hashed into a smaller value if overhead is an issue.

Invalidate reply. Same as above, except that the destination is set to the address of the local write instruction that caused the request and the context when the request was originated. This captures WAW edges.

Read miss serviced from memory. The source is set to NULL and the destination set to the local instruction and context when the miss happened. This captures information about instructions that typically do not read shared data.

5. BUG DETECTION

Bugaboo uses two basic methods of bug detection with context-aware communication graphs: *labeled* and *unlabeled*. In the labeled method, the programmer classifies each execution as *buggy* or *non-buggy*. Conversely, the unlabeled method is fully automatic, the programmer does not need to classify the executions. Below we describe each method and outline how to leverage BB-HW in deployment situations.

Bug detection using unlabeled runs. The most automatic way of using graphs for bug detection is to collect graphs from a number of executions then determine rare communication events that are likely to be the result of bugs. The crucial observation is that buggy communication is rare. Specifically, our debugging method based on unlabeled runs produces a ranked list of *code points*[2] for the programmer to examine. The rank for each code point, CP, is defined as: $rank_{CP} = \sum_{x \in X_{CP}} \frac{F_{CP,x}}{F_{CP,*}}$, where X_{CP} is the set of contexts in which CP executed, $F_{CP,x}$ is the number of runs in which code point CP executed in context x, and $F_{CP,*}$ is the total number of times CP executed regardless of context and across all runs. We sort the list in ascending order. In summary, code points that executed in rare contexts are ranked higher. In Section 6.2 we demonstrate that this method is very effective at detecting concurrency errors, in spite of a few irrelevant code points which get a high rank.

Bug detection using labeled runs. In this method, the programmer runs the application multiple times and labels each execution as *buggy* or *non-buggy*, depending on whether the bug manifested itself or not. This process can be assisted by testing tools that attempts to force bugs to happen [12, 14]. Once the runs are labeled, we produce a set of *bug-only* graphs. A bug-only graph is computed by taking a graph difference between the graph of each execution labeled as buggy and the union of all graphs obtained from executions labeled as *non-buggy*. We then apply the same exact ranking process used for the unlabeled method but over the set of bug-only graphs. By defining and ranking code points of a set of bug-only graphs, highly ranked code points are most likely related to the bug. We demonstrate experimentally (Section 6.2) that this technique locates bugs precisely with a small number of executions.

Our labeling process assumes that the programmer classifies each execution depending on the manifestation of only the specific bug being investigated. This means that the programmer does not need to know whether the whole execution of a program is correct, which is hard. The programmer just needs to know whether the bug in question manifested or not and classify accordingly. This reasonable assumption makes labeling simple and accurate.

Post-deployment bug detection using BB-HW. BB-HW's support for collecting context-aware communication graphs causes negligible performance degradation. Therefore we can use it in a deployment scenario to continuously monitor an execution and detect when an execution is likely to be buggy. When a deployed application is running in the field, a spare core in our system periodically collects graphs and processes them to detect whether it is likely that the deployed application is experiencing a new bug.

We provide this functionality using a combination of the two debugging methods just described. During testing, the developer collects context-aware communication graphs for all test cases. The resulting graphs are then unioned into the *testing graph*, which represents all executions observed during testing and can therefore be considered correct. The testing graph is deployed together with the application. For graphs collected periodically when the application is running in the field, the system takes the difference from the testing graph and applies the unlabeled method described earlier in this section. For the code points with a rank above a configurable threshold, the system sends the information back to the developer.

[2] A code point is a location in the source code, which can map to multiple static instructions in the binary.

6. EVALUATION

This evaluation aims to: (1) demonstrate that our debugging methods based on context-aware communication graphs detect bugs accurately, leading to few unnecessary code inspections; (2) characterize size and accuracy of our graphs; (3) characterize BB-SW; and (4) characterize the overheads in BB-HW, justifying our design choices.

6.1 Experimental Setup and Methodology

We evaluate BB-HW using a simulator based on Pin [11] and SESC [16]. The simulator models a 16-node multiprocessor, with 32KB 8-way associative L1 Caches (cf., Intel Core), MESI cache coherence protocol, our cache extensions, communication context registers, tracking protocol, 16k-entry communication table and software layer (traps).

We used three categories of workloads: full applications, bug kernels, and synthetic buggy code. Table 1 shows the workloads used. The full applications were chosen based on previous literature on bug detection [8, 10, 20]. To exercise buggy code in Apache, we enabled buffered logging, and used a custom script which launched 10 simultaneous requests for a static resource. For our experiments with MySQL, we enabled binary logging, and used the included sql-bench utility, modified to execute 50 instances of the test-insert benchmark in parallel. For PBZip2, we decompressed a bzip compressed text file containing a communication graph from our tool. For AGet we fetched a software archive from a remote server, and interrupted the download with the Unix interrupt signal. In AGet and PBZip2, we added Unix `usleep` calls to more frequently cause the bug to manifest itself. Our bug kernels were extracted from Mozilla and MySQL. They are 300-600 line extracts including buggy code from these applications. We used bug kernels to capture the essence of bugs, and make in-depth experimental analysis less cumbersome. This methodology has been used successfully in prior work in this area [9, 10, 20]. Finally, we used several synthetic bug benchmarks. Several of these were used in prior work on atomicity violation detection [9, 10], and we added a synthetic ordering violation bug.

6.2 Efficacy

We applied the labeled and unlabeled debugging methods described in Section 5. The output of those methods is a list of code points ordered by our ranking criterion. We measure the quality of the output by the number of non-buggy code points ranked higher than the bug, i.e., the *number of inspections* required before the bug is found. All results presented are averaged over 5 trials. For each labeled trial, we collected 25 buggy runs, and 25 non-buggy runs. For each unlabeled trial we used 25 runs, and ensured that at least

App. Class	Name	App. Version	Bug Type	Description
Synthetic	BankAcct	n/a	Atomicity Violation	Two threads try to update a bank account balance simultaneously, and an update is lost.
	CircularList	n/a	Atomicity Violation	Many threads remove elements from head of queue and append them to tail of queue. Lack of atomicity of remove/append leads to incorrect append order.
	LogAndSweep	n/a	Atomicity Violation	A log is written by many threads, and periodically flushed. Missing atomicity constraint leads to log corruption.
	MultiOrder	n/a	Ordering Violation	Two threads' repeated accesses to a shared variable must be interleaved. No code constraint enforces interleaving.
Bug Kernel	Moz-jsStr	Mozilla-0.9	Multi-Var. Atom. Vio.	To compute avg. string length, total number of strings and total string length are tracked. Non-atomic updates can permit these to become inconsistent
	Moz-jsInterp	Mozilla-0.8	Multi-Var. Atom. Vio.	Cache data structure is populated, and flag indicating cache occupancy is set. Lacking atomicity constraints, interleaving read may read flag while it is inconsistent with cache.
	Moz-macNetIO	Mozilla-0.9	Multi-Var. Atom. Vio.	Read of "valid" flag in conditional test and outcome of conditional can be interleaved, and data invalidated.
	Moz-TxtFrame	Mozilla-0.9	Multi-Var. Atom. Vio.	During update of buffer offset and buffer text length variables, inconsistent values can be read by interleaving read.
	MySQL-IdInit	MySQL-4.1.8	Ordering Violation	Query of database node ID should be ordered with assignment of node ID but absent ordering constraints lead to incorrect ID in query reply.
Full App.	MySQL-BinLog	MySQL-4.0.12	Atomicity Violation	Attempts to log data during log rotation do not properly handle log being closed, leading to unlogged database transactions.
	Apache-LogSz	Apache-2.0.48	Atomicity Violation	Concurrent updates to length of text in buffer can cause dropped update, leading to corruption of buffer. Can lead to crashes and log corruption.
	PBZip2-Order	PBZip2-0.9.1	Ordering Violation	Termination of worker thread loops is not ordered with deletion of pthread cond. var. data structure. Accesses to deleted cond. var. causes crash.
	Aget-MultVar	AGet-0.4	Multi-Var. Atom. Vio.	Value of shared var. should be consistent with # bytes written to output file. Lacking atomicity constraint permits read of inconsistent value of shared var. in signal handler.

Table 1: Bug workloads used to evaluate Bugaboo.

1 run was buggy. We justify the number of runs in Section 6.4. Table 2 lists each bug, whether we were able to detect it with and without context (Columns 2-3), and the number of code point and function inspections required to find the bug, using labeled and unlabeled methods in both BB-SW and BB-HW (Columns 4-9).

Overall, our results demonstrate that our technique accurately pin-points concurrency errors, even in very large software packages. In our experiments with labeled graphs, bugs were located with few inspections and in many cases, the bug was the *first code point reported*. Using unlabeled graphs, we saw comparable results, with little (if any) increase in required inspections. We now discuss the importance of context, the differences in accuracy between BB-SW and BB-HW and the effect of communication tracking granularity.

6.2.1 The Importance of Communication Context in Detecting Concurrency Bugs

Column 2 (*Detected without Ctx*) and Column 3 (*Detected with Ctx*) in Table 2 show that, with just one exception, the bugs evaluated can only be found with context information. The exception is an ordering violation bug: `MySQL-IdInit`. This bug can be detected without context because there is a pair of instructions that communicates only during buggy runs, irrespective of context. For the other bugs, without context, there were no communication edges in graphs from our experimental executions that occur only during buggy executions, making it impossible to detect them using differences of context-oblivious graphs.

Our dependence on context does not mean other techniques will not find these bugs. For example, AVIO can detect some of the atomicity violations that require context (e.g., `BankAccount`) because AVIO uses a heuristic specific to atomicity violations. We are not aware, however, of another approach that is able to detect the multivariable atomicity violations in Table 1.

6.2.2 Detecting Bugs With Labeled Communication Graphs

Columns 4, 6 and 8 in Table 2 show the number of code point inspections to find the bug using labeled graphs. Both BB-HW and BB-SW are able to detect all bugs, requiring few unnecessary inspections and, in some cases, none at all. The application requiring the most inspections was `MySQL-BinLog`, with approximately 34 (in 24 different functions), which is a reasonable number considering that the code consists of over one million lines. For `Apache-LogSz`, which has over 220k lines of code, the largest number of required inspections was 12 using BB-HW; using BB-SW, this drops to just 8.8 on average. For `Aget-MultVar`, a smaller application with less than 5k lines of code, there were never any code points ranked higher than the bug.

Generally speaking, comparing BB-HW with BB-SW (line granularity) shows a small decrease in the number inspections required for our full application workloads. The decrease is expected because graph collection in BB-HW is less precise, as it considers only cache-to-cache transfers as communication, whereas BB-SW considers all of memory. Comparing Columns 6 and 8 shows the effect of line-level tracking compared to word-level, which doesn't have a significant effect on the number of inspections required.

6.2.3 Detecting Bugs With Unlabeled Communication Graphs

Columns 5, 7 and 9 in Table 2 show the number of code point inspections required by our debugging method using unlabeled graphs. The results show there are typically few inspections required. In many cases (e.g., `Apache-LogSz` and `Moz-jsStr`) the number was larger than with the labeled method, which is intuitive, since the unlabeled method relies on less information (no labeling information from the user). Other cases show the opposite result: `PBZip2` and `MySQL-BinLog` actually got significantly better. We found the disparity surprising. It turns out that the cause is an artifact of our ranking function using *relative frequency* of contexts: the graph difference used to produce the *bug-only* graph in the labeled method made the context of the buggy code point less relatively rare, and therefore lower ranked. This is a somewhat undesirable effect of our ranking function. A virtue of using context-aware communication graphs, however, is that they can be used with any ranking metric or learning technique, and are not restricted to the technique proposed here.

Benchmark	Detected *without* Context	Detected *with* Context	# of Code Inspections To Find Bug					
			BB-HW		BB-SW *Line* Granularity		BB-SW *Word* Granularity	
			Labeled	Unlabeled	Labeled	Unlabeled	Labeled	Unlabeled
BankAcct	No	Yes	**1.4** (1.0)	**2.2** (1.2)	4.0 (1.0)	3.6 (1.2)	3.6 (1.4)	4.8 (1.2)
CircularList	No	Yes	**2.2** (1.2)	**2.0** (1.6)	1.2 (1.0)	1.0 (1.0)	2.6 (1.2)	2.0 (1.0)
LogAndSweep	No	Yes	**1.0** (1.0)	**1.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.0)
MultiOrder	No	Yes	**1.0** (1.0)	**1.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
Moz-jsStr	No	Yes	**1.8** (1.0)	**5.8** (3.2)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.4 (1.0)
Moz-jsInterp	No	Yes	**1.0** (1.0)	**1.4** (1.2)	2.6 (1.6)	3.8 (2.8)	1.8 (1.0)	2.8 (2.0)
Moz-macNetIO	No	Yes	**1.4** (1.0)	**2.8** (2.6)	5.0 (3.6)	2.4 (2.0)	3.0 (1.6)	4.8 (3.4)
Moz-TxtFrame	No	Yes	**1.0** (1.0)	**2.8** (2.4)	1.8 (1.4)	4.2 (2.4)	3.4 (1.0)	2.4 (2.2)
MySQL-IdInit	Yes	Yes	**1.0** (1.0)	**1.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
MySQL-BinLog	No	Yes	**34.0** (24.4)	**19.2** (11.0)	28.4 (19.2)	19.6 (14.8)	34.2 (21.2)	16.0 (10.8)
Apache-LogSz	No	Yes	**12.0** (10.6)	**80.2** (49.8)	8.8 (7.2)	125.4 (60.8)	13.2 (10.8)	142.0 (61.4)
PBZip2-Order	No	Yes	**14.5** (4.8)	**5.2** (1.6)	10.8 (2.6)	1.4 (1.0)	6.6 (2.0)	3.4 (1.6)
AGet-MultVar	No	Yes	**1.0** (1.0)	**1.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)

Table 2: Bug detection accuracy using Bugaboo. We report the number of code point inspections required before the corresponding bug was found, the number in parenthesis show the number of distinct functions. Note that one inspection indicates that zero irrelevant code points needed inspection, since the bug was found on the first. Results are averaged over five trials.

6.2.4 Sources of Irrelevant Reports

There are two main reasons irrelevant code points are sometimes highly ranked. First, nondeterministic multithreaded execution may lead to potentially rare, but correct communication. If sufficiently rare, or if only ever observed in buggy executions, these may be ranked highly, in spite of being correct. Labeling graphs largely mitigates this source of false positives. Second, because buggy behavior tends to be infrequent, when buggy code executes, the resultant communication context might also be infrequent with respect to subsequent communicating instructions not involved in bugs. This can lead to these instructions having a rare context and appearing as bugs in our ranking. This can be mitigated by using graphs from more runs.

6.3 Impact of Graph Collection Imprecision

As discussed in Section 4.2.2, we traded precision for lower complexity in BB-HW. Compared to whole-memory communication tracking at word granularity, there are two sources of imprecision in BB-HW: (1) communication tracking at a cache-line granularity; and (2) considering only cache-to-cache transfers as communication. We quantify imprecision by computing the number of distinct communicating code points that were present in the graph collected using whole-memory word-level (BB-SW word granularity) but *not* present in graphs collected using the lossy collections, viz., whole-memory line-level (BB-SW line granularity) and cache-to-cache line-level (BB-HW). Table 3 shows the results, which are consistent across the board. Column 2 shows that imprecision added by line tracking ranged from 17% to 27% and Column 3 shows that imprecision added by cache-to-cache and line tracking is about twice that.

App.	% Imprecision Introduced	
	Line-Aliasing	Aliasing + Evictions
AGet	16.67	33.33
Apache	25.57	59.09
MySQL	27.43	54.51
PBZip2	26.32	59.65

Table 3: Imprecision caused by line-level and cache-to-cache-only tracking of inter-thread communication.

Section 6.2 showed that neither source of imprecision affects Bugaboo's ability to accurately detect bugs. There are two reasons for that. First, we do not detect bugs based on *absolute* graph properties, but rather, by detecting graph anomalies — a property *relative* to the emergent communication invariants in a set of graphs. These anomalies manifest themselves even in graphs collected using line addresses, because aberrant communication events (potential bugs) still render themselves as rare edges, just as they do at word granularity. Second, cache evictions are not likely to have a significant impact on bug detection capability. The reason is that concurrency bugs typically manifest in tighter interleavings, which leads to short-range communication. The communication not captured due to cache evictions are ones which would have resulted from communication over a long range of instructions — long enough for data to be evicted from the cache — and are thus unlikely to be of any use in debugging.

6.4 Effect of Context Size

Table 4 shows how Bugaboo's bug detection ability varies with the context size. Note that some bugs can not be found with context lower than 4 (`PBZip2-Order`). For `MySQL-BinLog`, the number of inspections required to find the bug is high unless a longer context is used. For most applications, as context grows, the number of irrelevant inspections goes down, which is expected since more ordering information is available to distinguish memory accesses involved in bugs. We chose Bugaboo's default context size to be 5 because we wanted to favor better bug coverage and lower unnecessary inspections even if at the cost of increasing graph size.

Benchmark	None	1-Entry	2-Entry	3-Entry	4-Entry	5-Entry
BankAcct	—	2.0	2.0	2.4	3.4	3.6
CircularList	—	—	7.2	3.4	3.2	2.6
LogAndSweep	—	—	2.2	2.8	1.6	1.0
MultiOrder	—	—	2.8	1.0	1.0	1.0
Moz-jsStr	—	1.0	1.0	1.0	1.0	1.0
Moz-jsInterp	—	—	—	1.6	1.8	1.8
Moz-macNetIO	—	1.2	1.0	1.4	2.2	3.0
Moz-TxtFrame	—	—	3.4	2.8	2.8	3.4
MySQL-IdInit	1.0	1.0	1.0	1.0	1.0	1.0
MySQL-BinLog	—	—	522.6	128.8	60.0	34.2
Apache-LogSz	—	25.2	6.4	7.2	9.8	13.2
PBZip2-Order	—	—	—	—	6.6	6.6
AGet-MultVar	—	3.8	3.8	1.4	1.0	1.0

Table 4: Bug finding results for BB-SW (word) with different context sizes. Dash (—) indicates the bug was not found with the corresponding context size.

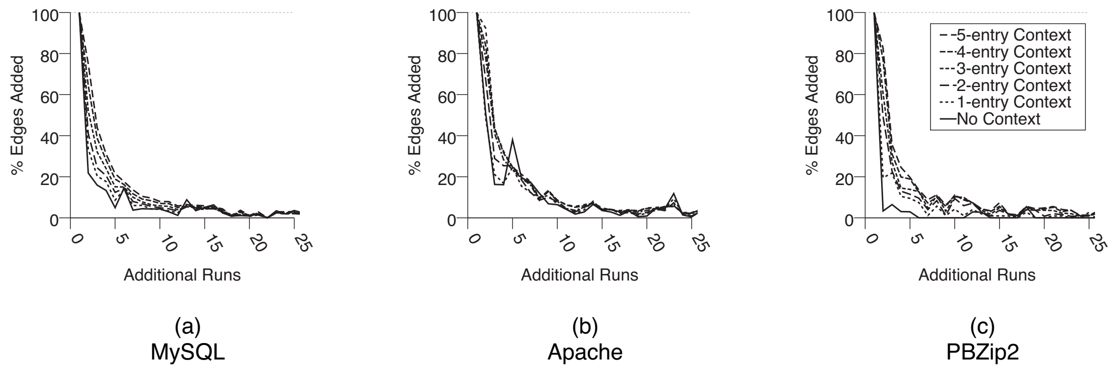


Figure 7: Graph convergence with increasing number of runs for MySQL (a), Apache (b), and PBZip2 (c).

We now show that after few program runs, the number of new communication events (edges) contributed by each additional run decreases rapidly, i.e., we obtain a *convergent* communication graph quickly. Figure 7 shows the number of new communication events contributed by each program run as a fraction of the total graph size after all prior runs. As expected, with longer contexts, more runs are necessary to reach a convergent graph.

Convergence is demonstrated in the sharp drop-off in new communication events, which reaches a flat bottom at around 10-15 runs. Our data shows that as the length of the context increases, the time to converge increases as well. The increase occurs because as context gets longer, each instruction potentially maps to a larger number of nodes. For a few runs, the fraction of edges added is greater for smaller context sizes. This apparent inversion is because graphs with longer contexts are larger, and the percent increase in edges contributed by the run is lower for larger graphs. These results show that the number of runs required to collect a convergent graph is proportionate to the length of the context, and that for any context size, very few runs are necessary. This justifies the choice of 25 runs for our evaluation, since it is sufficient for convergence.

6.5 Characterization

Our characterization has three goals: to understand the overheads in BB-HW; to assess the performance cost of BB-SW; and to measure the typical size of context-aware communication graphs. We did this characterization using the full applications from Table 1 as well as the PARSEC [1] benchmark suite, since synthetic bugs and bug kernels are not suitable for this characterization.

6.5.1 Overheads of BB-HW

The main sources of overhead in BB-HW are writes to cache line meta-data, writes to the CT, and traps to software when the CT is full. Column 5 in Table 5 shows that the number of meta-data updates is typically less than 200 per 10,000 memory operations, and is as few as 3 in 1 million memory operations. Meta-data updates only happen during cache misses (Columns 2, 3 and 4) and can be fully overlapped, being completely off the critical path. They therefore do not impose any performance cost. The number of CT updates (Column 6) is predominantly less than 35 per 10,000 memory operations. CT updates are done simultaneously with cache coherence transactions and the operation to perform is a simple FIFO buffer insertion, so it imposes negligible runtime overhead.

The most costly overhead is performing a software trap when the CT is full. Column 7 shows that traps happen just a few times per 10 million memory instructions. These events are sufficiently infrequent that their cost is amortized over the course of an execution. Moreover, it is possible to dedicate a spare core to read the communication table, reducing the cost further. During trap handling, writes to the CT are simply discarded. While this may result in a small number of missed graph edges, it enables uninterrupted execution during trap handling, making it effectively a zero-overhead operation.

Several applications stand out with higher costs: `freqmine`, `vips`, `MySQL-BinLog` and `x264`. These applications have 10 to 20 times more edges in their communication graphs and much higher cache miss rate (Columns 2, 3 and 4), indicating that they have more widespread and frequent communication. This ultimately leads to higher frequency traps to read out the communication table, but still, the frequency is only about 1 per million memory operations.

6.5.2 Overheads of BB-SW

Column 8 shows the slowdown caused by BB-SW compared to the application running natively, without any instrumentation (not under Pin). As expected, BB-SW causes significant performance degradation, since it has to monitor and take an action at every memory operation. The cost of the action varies depending on how frequently inter-thread communication occurs in the application. For some applications, such as `Apache-LogSz` and `AGet`, we saw tolerable slowdown of about 15x. For some applications (e.g., `Vips`), the cost was significantly higher, reaching three orders of magnitude in some cases. This is on par with popular dynamic analysis tools such as Valgrind [13]. Our focus in this work was not optimizing our software tool and we see direct ways of reducing overheads, e.g, by using more concurrent data structures.

6.5.3 Graph Sizes

Columns 9-12 show the size of communication graphs both with and without context. The size of graphs is a function of how widespread communication is within the application's code. The first noticeable trend is that context-aware communication graphs (Columns 11 and 12) are significantly larger than context-free communication graphs (Columns 9 and 10). This is expected, since instructions can execute in multiple contexts. For none of the applications did these graphs exceed 100k nodes and 200k edges in size. Using an un-optimized, sparse adjacency matrix representation, context-aware graphs never exceed 1 MB in size. The low storage overhead supports the feasibility of using BB-HW post-deployment, since doing so requires shipping the union of all graphs obtained during testing. Also, since graphs are small, differences obtained in the field can be transmitted back to the developer.

Benchmark	BB-HW						BB-SW	Graph Sizes			
	Rd. Ms. / 10k MOp	Wr. Ms. / 10k MOp	Coh. Ms. / 10k MOp	M-D Wr / 10k MOp	CT Wr. / 10k MOp	Traps / 10M MOp	Slow-down(x)	w/o Context # Nodes	w/o Context # Edges	w/ Context # Nodes	w/ Context # Edges
blackscholes	212.41	53.93	0.02	266.36	212.55	11.3	128	51	104	230	472
canneal	429.33	15.23	0.00	444.57	429.33	6.2	80	216	437	2025	4055
dedup	5.00	0.49	0.05	5.53	5.09	0.1	451	227	750	3784	11570
ferret	33.70	23.96	0.08	57.74	33.78	0.0	26	398	821	572	1216
fluidanimate	27.15	8.38	0.25	35.78	27.69	1.6	4623	284	831	15692	38570
freqmine	137.59	37.90	0.01	175.50	137.68	8.0	3845	1050	2228	41455	85142
swaptions	23.53	98.60	0.51	122.65	25.09	1.5	2151	168	676	5103	15633
vips	254.72	67.42	0.05	322.19	254.81	15.5	5025	1326	2942	56016	115178
x264	75.22	28.09	0.00	103.30	75.22	4.4	1260	2347	4799	68067	137071
AGet-MultVar	10.50	0.25	8.39	19.14	18.91	0.0	15	58	135	154	376
PBZip2-Order	0.02	0.00	0.00	0.03	0.02	0.0	19	59	145	208	451
Apache-LogSz	3.27	3.78	0.03	7.08	3.31	0.0	13	672	1361	1797	3635
MySQL-BinLog	129.15	32.85	0.18	162.18	129.41	6.3	166	1303	3271	20435	48861

Table 5: Characterization of BB-HW, BB-SW and communication graphs sizes.

6.6 Case Study: Configuration Error in `dedup`

After Bugaboo's development and initial evaluation, we decided to run it with a few PARSEC applications. We reconfigured the hashtable data structure used in `dedup` benchmark so that it was built with a configuration documented as unsafe for multithreaded execution — we enabled dynamic hash resizing. We then used *BB-SW* to collect graphs from 50 program runs, using the buggy configuration. We processed the collected graphs using our unlabeled processing technique. After examining just 6 code points, we discovered an atomicity violation which occurs when the buggy configuration is used, as well as a developer comment describing that the code was not safe if threading is enabled. While this was a documented configuration error, and not a new bug, we consider the ease with which we found the involved code a further validation of the effectiveness of our technique.

7. RELATED WORK

Debugging based on anomalous behavior has been explored extensively in the literature, including both static [2] and dynamic approaches [3, 6]. Liblit [7] has explored using statistical techniques for invariant-based bug detection in which he leverages execution diversity from deployed software. This prior work inspired us in thinking about using invariant-based techniques for concurrency bug detection.

AVIO [9] is an atomicity violation detection system that uses an invariant-based approach. AVIO uses a set of training runs to infer when memory accesses should not be interleaved; it then monitors dynamically when these invariants are violated. It proposes architecture extensions to decrease the performance cost of dynamic monitoring. AVIO is tailored to single-variable bugs only.

Interleaving-Constrained Shared-Memory MultiProcessor [20], which was concurrently developed with our work, is a bug avoidance technique based on building invariants during testing and then using architecture support that enforces these invariants at run-time. The invariants are encoded in sets of happens-before relationships between static memory instructions (called PSets), which is in effect a basic context-oblivious communication graph. Since PSets do not include any form of context, it fundamentally misses bugs that need context to be detected (e.g., multivariable ones), so our context-aware communication graphs enable detection of a superset of bugs that would lead to a PSet violation. Note that since PSets aims at bug avoidance, it needs more complex system and architecture support that can validate whether each memory access can proceed or not. Both AVIO and PSets only use labeled runs, so they are not fully automatic. Our unlabeled method, on the other hand, is fully automatic.

Other notable examples of architecture support for concurrency bug detection are HARD [21], which provides support for locking discipline violation [18] detection; Atom-Aid [10], which uses hardware signatures to detect atomicity violations; and ReEnact, which detects data races by leveraging speculative execution. The key differences of our proposed architecture extensions compared to previous proposals is: (1) we focus on a generic mechanism to build communication graphs; (2) we track global communication context; and (3) our mechanism is only activated during cache misses, so it incurs very low overhead.

Finally, DMTracker [5] also proposed to track communication in message-passing applications and use machine learning to detect potential bugs. DMTracker applies to a fundamentally different category of applications and types of bugs, but could potentially use our notion of context to increase their coverage and improve their results.

8. CONCLUSIONS

The approach to bug detection that we take in this paper is to collect communication graphs from multiple executions and identify graph anomalies that are likely the result of concurrency bugs. The main advantage of this approach over most prior work is that it is general. The key to graph-based bug detection is whether enough information is encoded in the graph.

In this paper we made the observation that basic context-oblivious graphs do not encode enough information to enable detection of many bugs. We addressed this issue by proposing *context-aware communication graphs*, a new type of graph that embeds access ordering information using communication contexts, which are easily obtained by monitoring inter-thread communication. Using these graphs, we developed Bugaboo, a comprehensive framework for concurrency bug detection that is able to accurately detect complex concurrency bugs (e.g., multivariable) with few irrelevant code inspections.

We describe two implementations of Bugaboo. One in software and one that uses hardware support to decrease the overhead of graph collection. The hardware support we propose is a set of modest architecture extensions to off-the-shelf multicore processors that efficiently keeps track of inter-thread communication and context. We leverage the fact that the architecture extensions do not lead to performance degradations and propose post-deployment use as well.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We thank Dan Grossman, Karin Strauss, Joseph Devietti, Tom Bergan,

Owen Anderson and Benjamin Wood for their invaluable feedback on the manuscript and insightful discussions. This work was supported in part by NSF under grants CNS-0720593 and CCF-0930512, and gifts from Intel, Microsoft and Google.

9. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[2] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, 2001.

[3] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *ICSE*, 2000.

[4] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, 2003.

[5] Q. Gao, F. Qin, and D. K. Panda. DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements. In *SC*, 2009.

[6] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE*, 2002.

[7] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

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

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

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

[11] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

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

[13] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

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

[15] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA*, 2003.

[16] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

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

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

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

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

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