2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

A Unified Approach for Localizing Non-deadlock Concurrency Bugs

Sangmin Park, Richard Vuduc, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
{*sangminp*|*richie*|*harrold*}*@cc.gatech.edu*

***Abstract*—This paper presents UNICORN, a new automated dynamic pattern-detection-based technique that finds and ranks problematic memory access patterns for non-deadlock concurrency bugs. UNICORN monitors pairs of memory accesses, combines the pairs into problematic patterns, and ranks the patterns by their suspiciousness scores. UNICORN detects significant classes of bug types, including order violations and both single-variable and multi-variable atomicity violations, which have been shown to be the most important classes of non-deadlock concurrency bugs. The paper also describes implementations of UNICORN in Java and C++, along with empirical evaluation using these implementations. The evaluation shows that UNICORN can effectively compute and rank the patterns that represent concurrency bugs, and perform computation and ranking with reasonable efficiency.**

I. INTRODUCTION

Concurrency is becoming more and more prevalent because of the increased availability of multi-core machines. A study performed at Microsoft reports that over 60% of developers face concurrency issues and most of the respondents handle concurrency bugs on a monthly basis [3]. Not only do concurrency bugs reduce the productivity of developers, but they can cause serious problems and disasters, such as the Northeastern blackout [19] and the Therac-25 accident [7], which were caused by race conditions.

Concurrency bugs are difficult to find because they occur in specific interleavings of memory-access sequences. The non-deterministic behavior of concurrent programs exacerbates finding these specific interleavings [14]. To monitor and investigate all memory accesses is practically impossible because the number of interleavings increases exponentially with the number of threads [15].

To detect such concurrency bugs, researchers have focused on finding those involving a single shared variable. Early work detects data races [2], [20] that, without synchronization, can result in problematic interleavings. Recently, researchers have investigated other important classes of concurrency bugs, such as order violations [17] and single-variable atomicity violations [10], [17], [18]. Although these techniques can successfully find bugs, their ability is limited to finding concurrency bugs involving a single variable.

Several existing techniques detect concurrency bugs involving multiple variables. Some techniques find multi-variable atomicity violations [4], [12], [23], but do not find some important classes of single-variable concurrency bugs, such as order violations. Other techniques report the existence of non-deadlock concurrency bugs involving both a single variable and multiple variables [6], [11], [13], [21]. However, these techniques do not report whether the bugs are order or atomicity violations, and thus, provide insufficient information to fully understand and fix the bug.

There are two main limitations of existing techniques. The first limitation is that existing techniques that handle multi-variable atomicity violations and provide sufficient information (e.g., access patterns) are not completely automated. These techniques require annotations of atomic regions [4], [23] or groups of memory locations [12] to find multi-variable problematic patterns only in the specified regions or groups. The second limitation is that there is no unified technique that detects both single- and multi-variable bugs together with sufficient information. A developer can use several existing tools, each of which detects a single type of concurrency bug with sufficient information, to detect all important classes of bugs. However, running several tools separately may increase testing time significantly, and understanding formats of reports from separate tools may require additional effort to find and fix the concurrency bugs.

To address the limitations of existing techniques, we developed, and present in this paper, UNICORN, a dynamic-pattern-detection technique that handles order, single-variable atomicity, and multi-variable atomicity violations[1] with a unified framework. UNICORN is based on the observation that problematic memory-access patterns representing concurrency bugs consist of several problematic memory-access pairs. UNICORN consists of three steps. In Step 1, UNICORN monitors memory-access pairs using a fixed-sized sliding-window mechanism, and records the program-execution outcome as either passing or failing. In Step 2, UNICORN combines memory-access pairs into problematic memory-access patterns using a second fixed-sized sliding-window mechanism for maintaining pairs. In Step 3, UNICORN computes the suspiciousness of the patterns and orders them in decreasing order of suspiciousness so that developers can quickly find the actual concurrency bugs from the bug report.

[1] According to a study of concurrency bug characteristics, these three types of violations are the most important classes of non-deadlock concurrency bugs [9].

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.17

IEEE computer society

UNICORN has several benefits over existing techniques. First, UNICORN is effective in detecting significant classes of bug types, including order violation and single-variable and multi-variable atomicity violations. Thus, UNICORN can provide more information to developers about the types of bugs than other techniques. Second, UNICORN reports patterns in order of suspiciousness, unlike other detectors that report benign and harmful results together without any ordering. Thus, developers save time in finding an actual bug using the bug report because they investigate harmful results before benign results. Finally, UNICORN integrates the detection of several classes of bugs and is implemented as a highly-automated single tool. In contrast, other techniques [6], [21] consist of several separate techniques implemented as single tools, which requires a developer to run all tools and to investigate all reports from the tools to find concurrency bugs that the techniques handle.

We also describe in this paper our implementations of UNICORN for both Java and C++, along with the results of empirical studies that we performed on a set of subjects to evaluate UNICORN. The first study investigates the effectiveness of UNICORN in detecting multi-variable atomicity violations, as well as in detecting order and single-variable atomicity violations. The study shows that, for our subjects, UNICORN ranks the problematic memory-access patterns as the most suspicious, and thus, effectively detects the concurrency bugs. The second study investigates the window size required to create the memory-access patterns that are combinations of memory-access pairs. The window size is a critical tuning parameter of our technique that affects the time and space overheads as well as the accuracy of the technique. The study shows that, for our subjects, our technique requires a small fixed-sized window to identify the most suspicious memory-access patterns. The third study investigates the efficiency of the technique. The study shows that, for our subjects, our technique has runtime overhead that is similar to related techniques.

The paper makes the following contributions:

- The presentation of a new technique that handles important classes of concurrency bugs: order violation, and single-variable and multiple-variable atomicity violations. To our knowledge, this is the first technique that targets and detects all these violations together with pattern and rank information.
- A description of implementations of the technique in Java and in C++.
- The results of a set of empirical studies that show the effectiveness of the technique in detecting concurrency bugs for multi-threaded programs. Importantly, problematic patterns that are directly related to the bug in all of our tests appeared at the top of the ranked list, suggesting that UNICORN will help developers locate bugs quickly.

Thread 1 (main)	**Thread 2 (worker)**
S_0: // pthread_join(worker); S_1: **mut** = NULL;	
	S_2: pthread_mutex_lock(**mut**);

Figure 1. Order violation.

II. CONCURRENCY BUGS

This section provides background for our technique. Section II-A introduces the notation we use, and Sections II-B to II-D define the concurrency violations that we address.

A. Notation

We denote a memory access to a shared variable by $b_{t,s}(x)$: b is the memory access type, either read (R) or write (W); t is the thread that performs the access; s is the program statement containing the access; and x is the shared variable. For example, $R_{1,S_1}(x)$ represents a read access to shared variable x in statement S_1 of thread 1.

Table I lists problematic memory-access patterns that represent the concurrency bugs in which we are interested. The first column shows the pattern ID. The second column shows the memory-access pattern using the shared-variable notation, $b_{t,s}(x)$. We discuss the details of the patterns later in this section. The third column shows the memory-access pairs that constitute the memory-access pattern. Note that the memory-access patterns are always decomposed into one or two memory-access pairs. We discuss how the pairs are used in our bug-detection technique in Section III. The fourth column gives a description of the pattern.

B. Order Violation

An *order violation* occurs when a pattern for order violation appears and leads to unintended program behavior. Patterns for order violation (Patterns P1 to P3) consist of two sequential thread accesses to a shared-memory location where at least one of the accesses is a write.

Figure 1 gives an example of an order violation.[2] The example consists of two threads, where the main thread (Thread 1) should wait at S_0 until the worker thread (Thread 2) finishes. If the main thread does not wait for the worker thread and deinitializes a shared variable `mut` with a null value at S_1, the program crashes with a null-pointer exception violation at S_2. In the example, an unintended order, $W_{1,S_1}(\texttt{mut})$-$R_{2,S_2}(\texttt{mut})$ (Pattern P2), is an order violation.

C. Single-Variable Atomicity Violation

A *single-variable atomicity violation* occurs when an unserializable interleaving pattern involving a single variable leads to unintended program behavior. An unserializable interleaving pattern is a sequence of concurrent memory accesses whose resulting state is not the same as that of sequential memory accesses. Patterns P4 to P8 are unserializable interleaving patterns involving a single variable [23].

[2] The examples are extracted from our subject programs. Figure 1 is from `PBZip2`, Figure 2 is from `MySQL-791`, and Figure 3 is from `MySQL-169`.

Table I
PROBLEMATIC MEMORY ACCESS PATTERNS OF THE BUGS THAT UNICORN ADDRESSES.

PID	Memory-Access Pattern	Memory-Access Pairs	Description
P1	$R_{1,S_i}(x)\ W_{2,S_2}(x)$	$R_{1,S_i}(x)\ W_{2,S_j}(x)$	An unexpected value is written.
P2	$W_{1,S_i}(x)\ R_{2,S_j}(x)$	$W_{1,S_i}(x)\ R_{2,S_j}(x)$	An unexpected value is read.
P3	$W_{1,S_i}(x)\ W_{2,S_j}(x)$	$W_{1,S_i}(x)\ W_{2,S_j}(x)$	The result of remote write is lost.
P4	$R_{1,S_i}(x)\ W_{2,S_j}(x)\ R_{1,S_k}(x)$	$R_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_j}(x)\ R_{1,S_k}(x)$	Two local reads expect to get the same value but do not.
P5	$W_{1,S_i}(x)\ W_{2,S_j}(x)\ R_{1,S_k}(x)$	$W_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_j}(x)\ R_{1,S_k}(x)$	A local read expects to get the result of a local write but does not.
P6	$W_{1,S_i}(x)\ R_{2,S_j}(x)\ W_{1,S_k}(x)$	$W_{1,S_i}(x)\ R_{2,S_j}(x)$, $R_{2,S_j}(x)\ W_{1,S_k}(x)$	A temporary result between local writes (in one thread) is not supposed to be seen to other thread but is.
P7	$R_{1,S_i}(x)\ W_{2,S_j}(x)\ W_{1,S_k}(x)$	$R_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_j}(x)\ W_{1,S_k}(x)$	The result of remote write is lost.
P8	$W_{1,S_i}(x)\ W_{2,S_j}(x)\ W_{1,S_k}(x)$	$W_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_j}(x)\ W_{1,S_k}(x)$	
P9	$W_{1,S_i}(x)\ W_{2,S_j}(x)\ W_{2,S_k}(y)\ W_{1,S_l}(y)$	$W_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_k}(y)\ W_{1,S_l}(y)$	The final values of x and y are inconsistent.
P10	$W_{1,S_i}(x)\ W_{2,S_j}(y)\ W_{2,S_k}(x)\ W_{1,S_l}(y)$	$W_{1,S_i}(x)\ W_{2,S_k}(x)$, $W_{2,S_j}(y)\ W_{1,S_l}(y)$	
P11	$W_{1,S_i}(x)\ W_{2,S_j}(y)\ W_{1,S_k}(y)\ W_{2,S_l}(x)$	$W_{1,S_i}(x)\ W_{2,S_l}(x)$, $W_{2,S_j}(y)\ W_{1,S_k}(y)$	
P12	$W_{1,S_i}(x)\ R_{2,S_j}(x)\ R_{2,S_k}(y)\ W_{1,S_l}(y)$	$W_{1,S_i}(x)\ R_{2,S_j}(x)$, $R_{2,S_k}(y)\ W_{1,S_l}(y)$	Observed read of x and y are inconsistent.
P13	$W_{1,S_i}(x)\ R_{2,S_j}(y)\ R_{2,S_k}(x)\ W_{1,S_l}(y)$	$W_{1,S_i}(x)\ R_{2,S_k}(x)$, $R_{2,S_j}(y)\ W_{1,S_l}(y)$	
P14	$R_{1,S_i}(x)\ W_{2,S_j}(x)\ W_{2,S_k}(y)\ R_{1,S_l}(y)$	$R_{1,S_i}(x)\ W_{2,S_j}(x)$, $W_{2,S_k}(y)\ R_{1,S_l}(y)$	
P15	$R_{1,S_i}(x)\ W_{2,S_j}(y)\ W_{2,S_k}(x)\ R_{1,S_l}(y)$	$R_{1,S_i}(x)\ W_{2,S_k}(x)$, $W_{2,S_j}(y)\ R_{1,S_l}(y)$	
P16	$R_{1,S_i}(x)\ W_{2,S_j}(y)\ R_{1,S_k}(y)\ W_{2,S_l}(x)$	$R_{1,S_i}(x)\ W_{2,S_l}(x)$, $W_{2,S_j}(y)\ R_{1,S_k}(y)$	
P17	$W_{1,S_i}(x)\ R_{2,S_j}(y)\ W_{1,S_k}(y)\ R_{2,S_l}(x)$	$W_{1,S_i}(x)\ R_{2,S_l}(x)$, $R_{2,S_j}(y)\ W_{1,S_k}(y)$	

Thread 1	Thread 2

void new_file (...) {
  saved_type = log_type;
S_1: log_type = CLOSED;
 	
S_3: log_type = saved_type;
}
	int mysql_insert (...) {
	
	S_2: if (log_type != CLOSED){
	  mysql_bin_logwrite(...);
	  }
	
	}

Figure 2. Single-variable atomicity violation.

Vaziri, Tip, and Dolby [23] proved that the unserializable interleaving patterns (P4 to P17) are complete, meaning that if an execution conforms to the patterns, the execution is not serializable.

Figure 2 shows an example of a single-variable atomicity violation. The program has two threads. Thread 1 closes an old file and creates a new file, during which `log_type` is temporarily set to CLOSED at S_1 and set to the original status at S_3. Thread 2 records a transaction into a log if `log_type` is not a CLOSED status at S_2. If `log_type` at S_2 reads a CLOSED status, which is set at S_1, Thread 2 mistakenly misses recording a transaction. Here, the interleaving, $W_{1,S_1}(\texttt{log_type})$-$R_{2,S_2}(\texttt{log_type})$-$W_{1,S_3}(\texttt{log_type})$ (Pattern P6), is a single-variable atomicity violation.

D. Multi-Variable Atomicity Violation

A *multi-variable atomicity violation* occurs when an unserializable interleaving pattern involving multiple variables leads to unintended program behavior. Patterns P9 to P17 are unserializable interleaving patterns involving multiple variables [23].

Thread 1	Thread 2

int generate_table (...) {
  lock (&LOCK_open);
S_1: TABLE.remove (...);
  unlock (&LOCK_open);
 	
  lock (&LOCK_log);
S_4: LOG.write (...);
  unlock (&LOCK_log);
}
	int mysql_insert (...) {
	
	  lock (&LOCK_open);
	S_2: TABLE.insert (...);
	  unlock (&LOCK_open);
	  lock (&LOCK_log);
	S_3: LOG.write (...);
	  unlock (&LOCK_log);
	
	}

Figure 3. Multi-variable atomicity violation.

Figure 3 shows an example of a multi-variable violation. The program has two variables, `TABLE` and `LOG`. The program needs to maintain the invariant that `LOG` records updates to `TABLE` in the order in which those updates occurred. For instance, if an entry is inserted into `TABLE`, the program should immediately log the transaction in `LOG`. Note that `TABLE` and `LOG` are individually protected by locks, but the two operations together are not. Consequently, the interleaving, $W_{1,S_1}(\texttt{TABLE})$-$W_{2,S_2}(\texttt{TABLE})$-$W_{2,S_3}(\texttt{LOG})$-$W_{1,S_4}(\texttt{LOG})$ (Pattern P9), causes `TABLE` and `LOG` to become out-of-sync.

III. TECHNIQUE

Our technique, which we call UNICORN, identifies the problematic access patterns, listed in Table I, that may cause concurrency bugs. Figure 4 depicts the technique, which inputs a concurrent program P and a test suite T, and

Figure 4. Overview of UNICORN; Sections III-A to III-C provide details.

consists of three steps. In Step 1, UNICORN executes P with T multiple times, collects memory-access pairs from each execution, and records the program-execution outcomes as passing or failing. In Step 2, for each execution, UNICORN combines the pairs into problematic memory-access patterns (listed in Table I). In Step 3, UNICORN computes the suspiciousness of the problematic memory-access patterns, and ranks the patterns in decreasing order by suspiciousness. Algorithm 1 gives the details of each step of the technique. Sections III-A to III-C discuss the algorithm in detail.

Algorithm 1: UNICORN Algorithm

Input : P: program, T: test suite, w: window size for pairs, m: number of executions
Output: $rankedPatterns$: a list of patterns ordered by suspiciousness
Data: $pairlist$: a list of pairs in an execution,
$orderedPairs$: an ordered list of pairs in an execution,
$pairs$: a map of a run id to $pairlist$,
$outcomes$: a map of a run id to Pass/Fail,
$patternset$: a set of patterns in an execution,
$allpatternset$: a set of patterns in all executions,
$patterns$: a map of a run id to a set of patterns,
$patternsToOutcome$: a map of a pattern with Pass/Fail to occurrence count,
$suspmap$: map of a pattern to its suspiciousness value

```
   // Step 1 (online): see Section III-A
1  P' = instrument(P)
2  for i ∈ {0..m-1} do
3  |   (pairlist, outcome) = execute(P', T)
4  |   pairs[i] = pairlist
5  |   outcomes[i] = outcome
6  end
   // Step 2 (offline): see Section III-B
7  for i ∈ {0..m-1} do
8  |   patternset = empty
9  |   pairlist = pairs[i]
10 |   for p ∈ pairlist do
11 |   |   patternset.add(p)
12 |   end
13 |   orderedPairs = orderPairs(pairlist)
14 |   n = size(orderedPairs)
15 |   for j ∈ {0..n-w-1} do
16 |   |   for k ∈ {j+1..j+w} do
17 |   |   |   p1 = orderedPairs[j]
18 |   |   |   p2 = orderedPairs[k]
19 |   |   |   if isPattern(p1, p2) then
20 |   |   |   |   pt = makePattern (p1, p2)
21 |   |   |   |   patternset.add(pt)
22 |   |   |   end
23 |   |   end
24 |   end
25 |   patterns[i] = patternset
26 end
   // Step 3 (offline): see Section III-C
27 for i ∈ {0..m-1} do
28 |   outcome = outcomes[i]
29 |   patternset = patterns[i]
30 |   for pt ∈ patternset do
31 |   |   patternsToOutcome[pt][outcomes] += 1
32 |   |   allpatternset.add(p)
33 |   end
34 end
35 for pt ∈ allpatternset do
36 |   pass = patternsToOutcome[pt][Pass]
37 |   fail = patternsToOutcome[pt][Fail]
38 |   susp = computeSusp (pass, fail)
39 |   suspmap[pt] = susp
40 end
41 rankedPatterns = rankPattern(suspmap)
42 return rankedPatterns
```

A. Step 1: Collect Pairs from Executions

Step 1 collects memory-access pairs from program executions, and does so *online*. This step inputs a concurrent program P and a test suite T, and outputs memory-access pairs for each execution as $pairs$, and program execution outcome as passing or failing for each execution as $outcomes$.

The algorithm first instruments shared read and write variables in P, and generates an instrumented program P' (line 1). The algorithm uses escape analysis to instrument only shared variables. Then, P' executes T m times to collect $pairs$ and $outcomes$ (lines 2–6). For each execution, the algorithm collects problematic memory-access pairs (Patterns P1 to P3) as $pairlist$, and program-execution outcome as $outcome$, determined by T (line 3). Then, the algorithm associates an execution i with $pairlist$ and $outcome$ in $pairs$ and $outcomes$, respectively (lines 4–5).

1) Collecting shared-memory access: For each shared-memory access encountered during execution of P' (line 3), the algorithm records five kinds of information: the memory location, the static source location, the memory-access type as read or write, the parent thread id, and the global access index of the memory access. The algorithm maintains a global-access index counter, so the algorithm issues a global-access index for every shared-memory access.

2) Sliding window for accesses: During execution of P' (line 3), the algorithm uses a fixed-sized sliding-window mechanism [17] to collect problematic memory-access pairs of the shared-memory accesses. Specifically, the algorithm maintains one window for each shared-memory location. When a shared-memory access occurs, the algorithm recognizes the memory location of the access and adds the access to the window of the same memory location. Inside a window, there are fixed-sized slots of accesses, and the algorithm updates and slides the slots with a shared-memory update policy [17]. When there is an attempt to add an access to a full window, the algorithm finds memory-access pairs involving the oldest access and discards the oldest access.

Figure 5(a) illustrates the sliding-window mechanism. The first column gives a serialized representation of three shared-memory accesses from a program execution. The second column shows access windows of size 3; for simplicity, we show windows only for variable x. The top window shows

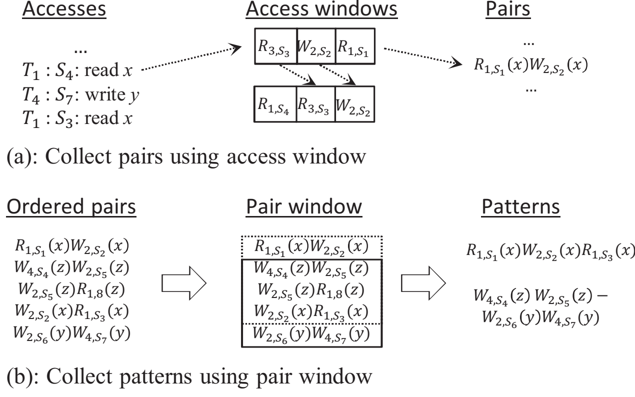

Figure 5. UNICORN uses two windows: (a) collects pairs from shared-memory accesses, and (b) collects patterns from memory-access pairs.

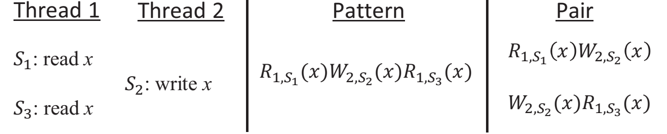

Figure 6. The technique is based on the observation that patterns consist of one or two pairs. The first column is an atomicity violation with $R_{1,S_1}(x)W_{2,S_2}(x)R_{1,S_3}(x)$, which is represented as a pattern in the second column, and is represented as two pairs in the third column.

the contents of the window for x before the execution of the first shared-memory access T_1:S_4:read x. When this first shared-memory access is executed, because the window is full, the algorithm finds pairs involving the oldest access R_{1,S_1}, generates the pair $R_{1,S_1}(x)W_{2,S_2}(x)$, which is shown in the third column, and discards this oldest access. The algorithm then updates the window by adding R_{1,S_4} as the newest access. The updated window is shown as the bottom window in the figure.

Because the algorithm maintains fixed-sized recent memory accesses, it may miss memory-access pairs where there are many intervening accesses by other threads. However, the empirical studies in our previous work [17] suggest that sliding-window techniques can often find buggy memory-access pairs with relatively small windows.

3) Pairs and patterns: The key idea behind the UNICORN technique is the collection of pairs from executions and the combination of the pairs offline to get the patterns. Unlike other techniques [10], [17], UNICORN does not collect patterns online because designing and maintaining online data structures to collect patterns for multi-variable concurrency bugs are complex to implement and may result in significant storage and time overheads. Therefore, UNICORN reconstructs patterns offline from the pairs collected online.

The intuition for this key idea is that problematic memory-access patterns can typically be captured by only one or two problematic memory-access pairs. Consider again Table I. Problematic memory-access patterns consist of two, three, or four memory accesses (second column), and they are represented by one or two memory-accesses pairs (third column). For example, the first column in Figure 6 shows an atomicity violation with $R_{1,S_1}(x)W_{2,S_2}(x)R_{1,S_3}(x)$, which is represented as a pattern in the second column, and is represented as two pairs in the third column.

This key idea explains the two advantages of the UNICORN technique. First, UNICORN focuses on patterns, which are sufficient for detecting multi-variable atomicity violations. Thus, developers can in turn understand the bug by inspecting the locations expressed in the pattern. Second, UNICORN maintains reasonable runtime overhead because it monitors only memory-access pairs. UNICORN performs its work offline to get complete bug information by combining pairs into patterns (See details in Step 2).

B. Step 2: Combine Pairs into Patterns

Step 2 of the algorithm combines memory-access pairs ($pairs$) into problematic memory-access patterns ($patterns$) (line 7), and does so *offline*. For each execution, a list of pairs, $pairlist$, is input and a set of patterns, $patternset$, is output. For patterns consisting of a pair (P1-P3), the algorithm adds all pairs in $pairlist$ to $patternset$ (lines 10–12). For patterns consisting of two pairs (P4-P17), the algorithm considers the combinations of two pairs in $pairlist$ and combines them when they represent a pattern (lines 13–25).

1) Sliding window for pairs: The algorithm uses a fixed-sized sliding-window mechanism to identify the combinations of the pairs efficiently. Step 1 uses a different window for each shared-memory location. However, Step 2 maintains only one window for the entire list of pairs, regardless of shared-memory locations, because Step 2 must combine patterns that involve two memory locations.

The algorithm first sorts $pairlist$ in increasing order of the global-access index of the first access in the pair (line 13). Then, the algorithm iterates over the ordered pairs in $orderedPairs$ (lines 15–25). The nested for loops iterate exactly n by k times, where n is the number of pairs in $orderedPairs$, and k is the window size. Note that k is a key parameter, as it controls both the accuracy and the overall complexity of our technique, thereby motivating Study 2 (Section IV-D). Thus, for any pair $p1$ and $p2$, the algorithm checks for a pattern with $p1$ and $p2$ when they are within the sliding window.

Figure 5(b) shows the way in which the sliding window mechanism works for pairs. The first column gives the collected pairs ordered by the global-access index. The second column shows how the sliding window with size 4 works. The sliding window has the first four patterns, shown within the dotted-line box, and the algorithm finds the $R_{1,S_1}(x)W_{2,S_2}(x)R_{1,S_3}(x)$ pattern within the window. Then, the window slides to the next four pairs, shown within the solid-line box, and the algorithm finds the $W_{4,S_4}(z)W_{2,S_5}(z)W_{2,S_6}(y)W_{4,S_7}(y)$ pattern within the window. The third column shows the two patterns.

Table II
APPEARANCES AND SUSPICIOUSNESS VALUES OF COLLECTED PAIRS (I1 TO I4) AND COMBINED PATTERNS (I5) IN SIX EXECUTIONS OF THE PROGRAM IN FIGURE 3.

ID	Pairs/Patterns	E1	E2	E3	E4	E5	E6	Susp
I1	$W_{1,S_1}(\texttt{TABLE})W_{2,S_2}(\texttt{TABLE})$	√		√		√	√	0.5
I2	$W_{2,S_3}(\texttt{LOG})W_{1,S_4}(\texttt{LOG})$		√		√	√	√	0.5
I3	$W_{2,S_2}(\texttt{TABLE})W_{1,S_1}(\texttt{TABLE})$		√		√			0.0
I4	$W_{1,S_4}(\texttt{LOG})W_{2,S_3}(\texttt{LOG})$	√		√				0.0
I5	$W_{1,S_1}(\texttt{TABLE})W_{2,S_2}(\texttt{TABLE})$ $W_{2,S_3}(\texttt{LOG})W_{1,S_4}(\texttt{LOG})$					√	√	1.0
		P	P	P	P	F	F	

2) Check for a pattern: Given two pairs, Step 2 checks whether the pairs represent a pattern (P4-P17) (line 19). First, the algorithm checks whether the pairs belong to the same thread; if not, they cannot be combined. Then, the algorithm checks whether the pairs are in the form of P4 to P17; if so, the algorithm creates a pattern, and adds the pattern to $patterns$ (lines 20–21).

3) Example: Table II shows how the algorithm works for Steps 1 and 2. The table shows appearances of collected pairs and combined patterns for six executions of the program in Figure 3. The first column shows the IDs. The second column shows the collected pairs and combined patterns. The third (E1) to eighth (E6) columns show appearances of the pairs and patterns in six program executions. The ninth column is the suspiciousness values, which will be explained in Section III-C. I1–I4 are collected pairs with two memory accesses, and I5 is a combined pattern with four memory accesses. The final row shows program execution outcomes as passing (P) or failing (F). Now, consider each execution from E1 to E6. In E1, Step 1 detects and records two pairs, $W_{1,S_1}(\texttt{TABLE})W_{2,S_2}(\texttt{TABLE})$ and $W_{1,S_4}(\texttt{LOG})W_{2,S_3}(\texttt{LOG})$. Then, Step 2 tries to create a new pattern consisting of the two pairs, so it checks whether the two pairs represent a new pattern. Step 2 finds that the two pairs are from the same threads, but the combination of the two pairs (i.e., $W_{1,S_1}(\texttt{TABLE})W_{2,S_2}(\texttt{TABLE})W_{1,S_4}(\texttt{LOG})$ $W_{2,S_3}(\texttt{LOG})$, or $W_{1,S_4}(\texttt{LOG})W_{2,S_3}(\texttt{LOG})W_{1,S_1}(\texttt{TABLE})$ $W_{2,S_2}(\texttt{TABLE})$) are not listed as a pattern in Table I. Thus, Step 2 cannot create new patterns from E1. In the same way, Step 2 cannot create any new pattern from E2 to E4. However, Step 2 can create a pattern I5, $W_{1,S_1}(\texttt{TABLE})W_{2,S_2}(\texttt{TABLE})$ $W_{2,S_3}(\texttt{LOG})W_{1,S_4}(\texttt{LOG})$, in E5 and E6.

C. Step 3: Rank Patterns

Step 3 of the algorithm computes a rank for each pattern, and presents the result to a developer. Step 3 inputs combined patterns $patterns$ and program-execution outcome $outcomes$, and outputs the ordered list of patterns in decreasing order of suspiciousness as $rankedPatterns$.

The algorithm first associates patterns with program-execution outcome; $patternsToOutcome$ records the number of occurrences of a pattern in passing and failing executions (lines 27–34). Then, the algorithm computes the suspiciousness of each pattern with its number of occurrences in passing and failing executions, and records the result in $suspmap$ (lines 35–40). Finally, the algorithm ranks the patterns in decreasing order of suspiciousness, and returns the result (lines 41–42).

Our technique uses the Jaccard index [17] to compute suspiciousness of the patterns as follows (line 39):[3]

$$\text{suspiciousness}_J(s) = \frac{\text{failed}(s)}{\text{totalfailed} + \text{passed}(s)} \qquad (1)$$

Consider again the example in Table II. The total number of failures is two. I1 appears in two failing and two passing executions. Thus, the suspiciousness of I1 is 0.5 using Formula (1). However, the suspiciousness of I5 is 1.0 because the pattern appears only in failing executions. Indeed, I5 is the actual multi-variable atomicity violation.

IV. EMPIRICAL STUDIES

To evaluate the UNICORN technique, we implemented a prototype for both C++ and Java, and used the prototype to perform empirical studies with a number of C++ and Java subjects. Section IV-A describes the implementation, and Section IV-B describes the empirical setup. Then, Sections IV-C to IV-E present the studies. Finally, Section IV-F discusses the threats to the validity of the studies.

A. Implementation

We implemented two modules for our technique (see Section III): one module for Step 1, and the other module for Steps 2 and 3. For Step 1, we implemented the module in Java and C++. The module takes a program written in Java or C++, and instruments shared variable accesses of the program, so that the instrumented program will dynamically output memory-access pair information.

For Java, we modified the FALCON framework to collect memory-access pairs [17]. FALCON uses static escape analysis to determine possibly thread-escaping variables, and instruments read and write accesses of the shared variables in the program. We made two changes to FALCON. First, we modified it to maintain sliding windows to detect memory-access pairs, whereas the original framework detects both pairs and triples. Second, we added a global access index counter, which is updated for every shared-memory access (recall Section III-A).

For C++, we created a module to instrument the subject programs statically using the LLVM analysis framework.[4] The module performs dynamic thread-escape analysis to find possibly shared variables in the program. Then, the module instruments the escaping load and store instructions and memory operations, such as `malloc`, `memcpy`, and

[3] In the equation, s indicates a pattern, failed(s) is the number occurrences of s in failing executions, passed(s) is the number occurrences of s in passing executions, and totalfailed indicates the number of total failing executions.

[4] http://www.llvm.org

Table III
SUBJECTS USED IN THE PAPER.

Type	Program	%Failed	PID	LOC	Type of Bug
C++ extracted	TimerThread	14.4	P2	68	Order
	LoadScript	49.8	P5	110	Single-variable atomicity
	MysqlLog	2.8	P6	89	Single-variable atomicity
	JsString	1.4	P12	95	Multi-variable atomicity
	MysqlDelete	3.8	P9	103	Multi-variable atomicity
	MysqlSlave	0.4	P14	94	Multi-variable atomicity
C++ full	PBZip2	2.8	P2	2K	Order
	Mysql-791	64.0	P6	372K	Single-variable atomicity
	AGet	49.0	P9	1.2K	Multi-variable atomicity
	Mysql-169	63.0	P9	331K	Multi-variable atomicity
Java library	StringBuffer	22.3	P14	1.4K	Multi-variable atomicity
	Vector	7.6	P14	9.5K	Multi-variable atomicity

`memset`. During runtime of the execution of the instrumented programs, the module uses the dynamic library interposition method [5] to collect memory-access pairs.

The Java and C++ modules for Step 1 generate output files in XML format. These files contain memory-access pairs and the program execution outcome for each execution. For Steps 2 and 3, we implemented a module in Java that uses the XML files generated in Step 1. For Step 2, the module reads each XML file that has memory-access pair information and generates a new XML file that has memory-access pattern information. For Step 3, the module inputs the new XML files generated in Step 2, computes suspiciousness for each pattern, and generates the result.

B. Empirical Setup

Table III lists the subject programs we used for our studies [24]. The first column shows the types of the subject programs in three categories: C++ extracted programs, which are extracted buggy parts of program code from Mozilla and MySQL; C++ full programs, which are full applications without any simplification; and Java library programs, which are extracted classes from the Java 1.4 library. The second column shows the name of the subject program. The third column shows the failure rate that we observed empirically with our test cases. The fourth column shows the pattern ID from Table I. The fifth and sixth columns list the size of the program in lines of code and the type of the concurrency bug, respectively.

We created test cases for the subjects. For the C++ extracted programs, we created test cases to call the extracted buggy parts concurrently and to determine program execution outcome. For C++ full programs, we provided inputs to the subjects that can trigger concurrency bugs. `PBZip2` is an application that compresses files using the bzip2 algorithm with parallel threads. We call `PBZip2` to compress a large file with a number of threads. `MySQL` is a widely used open-source database application. We concurrently call several queries that can trigger bugs in the database server. `AGet` is a multi-threaded download accelerator. We call `AGet` to download a large file from the Internet, and send a unix signal to stop the program and to trigger the concurrency bugs in the program. For Java library programs, we created a test case that create objects of the Java library classes, and execute the methods of the objects concurrently.

We ran our studies on a linux desktop with i5 2.8 GHz CPU and 8 GB of memory. We used gcc 4.4 and Java 1.5.

C. Study 1: Effectiveness

The goal of this study is to investigate how well UNICORN ranks the pattern of the actual bug. To do this, we set the window size for Step 1 of the algorithm to 5, and collected memory-access pairs in the C++ and Java subjects. To increase the probability of program failures, we inserted random artificial delays into the subject programs [1]. Then, we executed each subject program 100 times. We set the window size for Step 2 of the algorithm to 100.

Columns 2–8 in Table IV show the results of the study. Columns 2, 3, and 4 report suspiciousness values of the first memory-access pair, the second memory-access pair, and the memory-access pattern of the actual bug, respectively. Column 5 reports the rank of the pattern that represents the actual bug. Column 6 reports the number of patterns that rank first. Columns 7 and 8 report the number of collected pairs and the combined patterns collected for the study. For example, `TimerThread` has an order violation of $W_{1,S_i}(x)R_{2,S_j}(x)$ pattern, so we report the suspiciousness of the pair in Columns 2 and 4, but we do not report any value in Column 3. There are three patterns that rank first in the report. We investigated the three patterns manually, and found that one of the three patterns is an order violation, and two other patterns are atomicity-violation patterns, which always appear with the actual order violation. There are six reported pairs and seven reported patterns.

We make several observations about the results of the study that are summarized in Table IV under Study 1. First, the patterns at the first rank are all actual bugs for all our subject programs. Thus, for these subjects, our technique is effective in finding non-deadlock concurrency bugs, and developers can investigate only top-ranked patterns to find the actual bug. Second, some pairs that consist of the actual bug have low suspiciousness, but the pattern that manifests the actual bug has a high suspiciousness score. Thus, our pattern-combination technique is effective. For example, consider `MysqlSlave`. The suspiciousness values of the memory-access pairs are low, because they appear in both passing and failing executions. However, the two pairs appear together only in failing executions, and the suspiciousness of combined pattern is 1.0. Third, UNICORN sometimes reports several bugs at the top rank (e.g., Mysql-169 reports seven patterns with rank 1). From manual inspection, we found that all patterns ranked at the top are directly related to the actual bug. In future work, we plan to develop a new technique to group together such patterns that represent the same root cause of a concurrency bug.

Our results show that UNICORN successfully finds all three important classes of non-deadlock concurrency bugs

Table IV
RESULTS OF STUDY 1 (EFFECTIVENESS), STUDY 2 (PAIR WINDOW DISTANCE), AND STUDY 3 (EFFICIENCY).

Program	Study 1 (Effectiveness)							Study 2 (Window)		Study 3 (Efficiency)	
	Susp (Pair 1)	Susp (Pair 2)	Susp (Pattern)	Rank (Pattern)	# Rank 1	# Pairs	# Patterns	Median distance	Maximum distance	Slowdown	Time (Steps 2–3)
TimerThread	1.0	-	1.0	1	3	6	7	-	-	3.14×	<1s
LoadScript	0.86	1.0	1.0	1	2	3	4	2	2	9.54×	<1s
MysqlLog	0.02	1.0	1.0	1	2	4	5	2	2	4.95×	<1s
JsString	0.04	1.0	1.0	1	3	5	8	3	3	12.4×	<1s
MysqlDelete	0.04	0.27	1.0	1	1	7	1	2	2	3.48×	<1s
MysqlSlave	0.01	0.66	1.0	1	1	6	11	2	2	3.26×	<1s
PBZip2	0.42	-	0.42	1	6	59	333	-	-	1.18×	4s
Mysql-791	1.0	0.64	1.0	1	4	1082	10936	18	27	36.4×	57s
AGet	0.51	0.50	0.70	1	1	37	94	2	3	1.20×	2s
Mysql-169	0.63	0.63	1.0	1	7	894	9051	41	41	18.5×	48s
StringBuffer	0.58	1.0	1.0	1	3	8	18	4	5	1.23×	3s
Vector	1.0	1.0	1.0	1	4	11	25	2	2	3.54×	3s

Table V
COMPARISON TO RELATED TECHNIQUES.

Techniques	Order Violation	S-Atom. Violation	M-Atom. Violation
AVIO [10], CTrigger [18], PENELOPE [22], Falcon [17]	△	√	
ColorSafe [12], AtomTracker [16], Atomic-Set [4], [23], LifeTx [25]		√	√
CCI [6], Bugaboo [11], Recon [13], DefUse [21]	△[5]	△	△
Unicorn	√	√	√

for our subject programs. By contrast, prior approaches are limited either to (a) finding just the atomicity violations (e.g., [10], [12]), or (b) not providing pattern-based reports (e.g., [11], [21]). Thus, they may give the developer only partial information about which accesses are involved in the bug, whereas a full pattern, which our technique reports, reveals the bug directly. We discuss related work in more detail in Section V using Table V.

D. Study 2: Pair window distance

The goal of this study is to investigate the practicality of the window size used for computing pairs in Step 2 of the UNICORN algorithm (see Section III-B). Recall that decreasing the window size might reduce runtime overhead at the cost of reduced accuracy. To do this, we first executed the instrumented programs and computed the distance between the two pairs that constitute the actual concurrency bugs. We then computed the median and maximum distances of the pairs from the ordered list of all pairs.[6] The median lets us estimate the window size that will give a 50% chance of observing the buggy pattern; the difference between the median and maximum hints at the variance of the distance across test runs.

Columns 9–10 in Table IV show the median and maximum distances between the two pairs that constitute the actual concurrency bug. The distance includes the two pairs, so the minimum distance will be 2. For example, both the median and maximum distances between the two pairs of `LoadScript` are 2, so the first pair always precedes the second pair with no pairs in between.

We make three observations from the results. First, the pairs that constitute the concurrency bug appear close together with a maximum distance of 41, which occurs in `Mysql-169`. Thus, with a window size of 100, which we used in Study 1, we can find the problematic patterns. The distance between the first and the last accesses of the bug in `Mysql-169` is nearly 20,000 instructions long [12]. Thus, were we to use instruction-level monitoring techniques as used in other work [12], we would need to maintain a much larger window. In contrast, we can maintain a window with size 41 pairs using our technique, because we effectively operate at the source level rather than the instruction level.

A second observation concerns how the window size grows as programs become more complex, longer running, or more multithreaded. Column 7 shows that the subjects other than `mysql` required recording between 3–59 pairs, whereas the `mysql` subjects required 894 or more, which represents a 10×–100× difference. Yet, Column 9 shows that the subjects other than `mysql` had a median distance of at most 4 compared to at most 41 for `mysql`, which is roughly a 10× difference. That is, the distance (and therefore optimal window size) may not need to grow more than linearly with the number of pairs. This observation suggests our technique has the potential to scale with more complex programs.

Third, the maximum distance is always less than two times the median distance. Thus, we do not need to maintain a large fixed-sized window for outlier pairs that appear very far from each other, but we can maintain a small fixed-sized pair window to create the actual bug pattern from pairs.

E. Study 3: Efficiency

The goal of this study is to investigate the efficiency of the UNICORN framework in terms of the execution time overhead. To do this, we measured and compared execution times for uninstrumented and instrumented versions of our subject programs, and measured the execution time of the offline module used in Steps 2 and 3 of Algorithm 1.

[5] △ implies that the techniques neither distinguish the types of concurrency bugs nor show the details of the causes of each concurrency bug.

[6] We do not investigate the distance between the pairs when the bug is an order violation because order violations consist of only one pair.

Columns 11–12 in Table IV show the results of the efficiency study. Column 11 reports the slowdown factor from the uninstrumented version to the instrumented version of the program. Column 12 reports the execution time for the statistical module for Steps 2 and 3 of the algorithm. For example, the runtime of `Mysql-791` increases by 36.4×, and the execution time of the statistical module is 57s.

In terms of the runtime overhead, UNICORN has comparable overhead to the previous techniques [17], [21], [24]. There are noticeable differences in the time overhead between the classes of the subject programs (Column 10): C++ extracted programs have 6.13× overhead, C++ full programs have 14.32× overhead, and Java library programs have 2.3× average overhead. For C++ programs, previous related works [21], [24] have captured memory-access pairs as UNICORN does, but with an average increase in execution time of 100× and 20×. For Java programs, the most closely related FALCON work [17] incurs an average time increase for the same subjects of 2.3×. The techniques that detect concurrency bugs by instrumenting and monitoring all shared-variable accesses should also have comparable time overheads. UNICORN online execution time overheads are comparable. In terms of the offline execution time overhead, UNICORN incurs reasonable overhead was reasonable for our subjects, taking one minute or less in all cases.

F. Threats to validity

There several main threats to validity of the studies. Threats to internal validity include the empirical setup for our empirical evaluation, and in particular the test suite we used for the study. We did not measure the coverage or diversity of the interleavings of the test suite. However, to collect passing and failing executions with different thread schedules, we ran the program many times, which is similar to what would be done in practice. In addition, to increase the probability of the failures, we use artificial delay injection technique.

Threats to external validity limit the extent to which our results will generalize to other kinds of concurrent programs. Because of the lack of large and widely accepted bug benchmark suites, this threat is actually common to all prior work in this area. To mitigate this problem, our benchmark suite includes most of the benchmarks used in other work, and furthermore, includes both Java and C++ programs.

Threats to construct validity include the way developers find and fix bugs. The metric we use assumes that developers inspect the program with the memory access patterns in the rank order, and stop inspection when they reach the fault. Although this process may not be the real debugging situation, this approach to assessing effectiveness is used in many previous studies for statistical fault localization.

V. RELATED WORK

There is much research in finding concurrency bugs. In this section, we discuss existing techniques that find concurrency bugs and compare UNICORN to them. Table V qualitatively compares UNICORN to the related work in terms of concurrency bug-detection ability.

Much prior research on concurrency bug detection has focused on bugs involving a single variable (the second row in Table V). Some techniques focus on finding single-variable atomicity violations. AVIO [10] collects benign patterns for single-variable atomicity violations from passing executions, and finds and reports patterns that are not included in the benign patterns from failing executions. CTrigger [18] and PENELOPE [22] statically examine possible atomicity violations from an execution trace, and dynamically verify the violations. FALCON [17] is one of the first techniques that can find both order violations and single-variable atomicity violations. Like UNICORN, FALCON collects memory-access patterns dynamically and computes the patterns with suspiciousness scores with the same overhead as UNICORN. However, the bug-detection ability of all these techniques, including FALCON, is limited to concurrency bugs involving a single variable, whereas UNICORN finds both single-variable and multi-variable concurrency bugs.

There are several existing techniques for detecting multi-variable atomicity violations (the third row in Table V). Some techniques require programmer annotations, either to specify the atomic regions that should be protected from interleavings [4], [23] or to specify groups of memory accesses in which the memory accesses should be serialized [12]. More recent techniques [16], [25] automatically infer atomic regions in passing program executions, and find or avoid multi-variable atomicity violations by monitoring memory operations on atomic regions. Compared to these approaches, UNICORN has two advantages. First, UNICORN does not require any annotations. Second, UNICORN also handles order violation, whereas the other techniques are limited to only atomicity violations.

There are several existing techniques for detecting non-deadlock concurrency bugs without identifying the root cause of the bug (the fourth row in Table V). CCI [6] monitors and samples shared memory locations and reports the likely buggy location using statistical methods. Bugaboo [11] collects communication graphs that contain a list of memory locations between threads and reports the graph with suspiciousness ranking. Recon [13] extends Bugaboo to reconstruct the buggy source and sink locations of two different threads from the communication graphs. Defuse [21] monitors the memory-access pairs between threads and report the most suspicious pairs as possible concurrency bugs. UNICORN differs from these techniques in several ways. First, these techniques report bugs without the root causes, so developers need to manually find the root cause to check whether the bug is an order violation or an atomicity violation. Second, these techniques may not report all important locations of the bug, so developers may not fully understand and fix the bug using the bug report. For ex-

ample, multi-variable atomicity violations require two pairs of memory accesses, but techniques, such as Defuse [21], report only one pair of accesses. In contrast, UNICORN reports the detailed bug information with patterns that show the cause of the bug and the complete pairs of memory accesses.

VI. CONCLUSION AND FUTURE WORK

This paper presents UNICORN, the first unified technique that detects and ranks non-deadlock concurrency bugs using patterns. UNICORN collects memory-access pairs dynamically, and combines memory-access patterns from memory-access pairs offline. The technique has manageable runtime overhead, comparable to other techniques, while extending detection ability from single-variable concurrency bugs to both single- and multi-variable concurrency bugs. Our empirical studies show that UNICORN is effective and efficient for a suite of C++ and Java subjects.

Although the results of our empirical studies illustrate that UNICORN is promising, there are several areas of future work that will improve it. First, current approaches, including our technique, show significant slowdowns over uninstrumented programs. Thus, we need to develop a scheme to efficiently monitor memory accesses during program execution. Techniques such as sampling [6], annotation [12], and static analysis [8] could ameliorate the overheads in our technique. Second, our findings show that a concurrency bug may be represented with several patterns at the top rank, meaning developers will need to inspect the patterns at the top and manually cluster them into the same bug. Thus, we need to develop more succinct and precise reports to developers. We expect that calling context information could help address this issue. Third, similar to dynamic fault-localization techniques, out technique uses test suites. We did not investigate the effectiveness of fault localization different test suites. In our future work, we will investigate the quality of test suites, and provide a guidance for creating high-quality test suites.

ACKNOWLEDGEMENTS

This research was supported in part by NSF CCF-0541048, CCF-0725202, CCF-1116210, and CAREER Award 0953100, and IBM Software Quality Innovation Award to Georgia Tech. The reviewers provided many helpful suggestions that improved the paper's presentation.

REFERENCES

[1] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," in *Java Grande-ISCOPE*, 2001.

[2] C. Flanagan and S. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI*, 2009, pp. 121–133.

[3] P. Godefroid and N. Nagappan, "Concurrency at microsoft: An exploratory survey," in *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[4] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *ICSE*, 2008, pp. 231–240.

[5] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *FSE*, 2010, pp. 57–66.

[6] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *OOPSLA*, 2010, pp. 241–255.

[7] N. G. Leveson, "SafeWare / system safety and computers," 1995.

[8] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: automatically inferring multi-variable access correlations and. detecting related semantic and concurrency bugs," in *SOSP*, 2007, pp. 103–116.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes a comprehensive study on real world concurrency bug characteristics," *SIGARCH Comput. Archit. News*, no. 1, pp. 329–339, 2008.

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006, pp. 37–48.

[11] B. Lucia and L. Ceze, "Finding concurrency bugs with Context-Aware communication graphs," in *MICRO*, 2009, pp. 553–563.

[12] B. Lucia, L. Ceze, and K. Strauss, "ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violation," in *ISCA*, 2010, pp. 222–233.

[13] B. Lucia, B. P. Wood, and L. Ceze, "Isolating and understanding concurrency errors using reconstructed execution fragments," in *PLDI*, 2011, pp. 378–388.

[14] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comp. Surveys*, no. 4, pp. 593–622, 1989.

[15] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *SIGPLAN Not.*, no. 6, pp. 446–455, 2007.

[16] A. Muzahid, N. Otsuki, and J. Torrellas, "AtomTracker: a comprehensive approach to atomic region inference and violation detection," in *MICRO*, 2010, pp. 287–297.

[17] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *ICSE*, 2010, pp. 245–254.

[18] S. Park, S. Lu, and Y. Zhou, "CTrigger: exposing atomicity violation bugs from their hiding places," in *ASPLOS*, 2009.

[19] K. Poulsen, "Tracking the blackout bug," *SecurityFocus*, 2004.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," *ACM Trans. Comput. Syst.*, no. 4, pp. 391–411, 1997.

[21] 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, pp. 160–174.

[22] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: weaving threads to expose atomicity violations," in *FSE*, 2010, pp. 37–46.

[23] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an Object-Oriented language," in *POPL*, 2006, pp. 334–345.

[24] J. Yu and S. Narayanasamy, "A case for an interleaving constrained Shared-Memory Multi-Processor," in *ISCA*, 2009.

[25] ——, "Tolerating concurrency bugs using transactions as lifeguards," in *MICRO*, 2010, pp. 263–274.