Griffin: Grouping Suspicious Memory-Access Patterns to Improve Understanding of Concurrency Bugs

Sangmin Park
College of Computing
Georgia Institute of Technology
sangminp@cc.gatech.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
harrold@cc.gatech.edu

Richard Vuduc
College of Computing
Georgia Institute of Technology
richie@cc.gatech.edu

ABSTRACT

This paper presents Griffin, a new fault-comprehension technique. Griffin provides a way to explain concurrency bugs using additional information over existing fault-localization techniques, and thus, bridges the gap between fault-localization and fault-fixing techniques. Griffin inputs a list of memory-access patterns and a coverage matrix, groups those patterns responsible for the same concurrency bug, and outputs the grouped patterns along with suspicious methods and bug graphs. Griffin is the first technique that handles multiple concurrency bugs. This paper also describes the implementation of Griffin in Java and C++, and shows the empirical evaluation of Griffin on a set of subjects. The results show that, for our subjects, Griffin clusters failing executions and memory-access patterns for the same bug with few false positives, provides suspicious methods that contain the locations to be fixed, and runs efficiently.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Monitors, and Tracing*

General Terms

Algorithms, Reliability

Keywords

Concurrency, Debugging, Fault Comprehension

1. INTRODUCTION

The widespread deployment of concurrent systems has necessitated an increase in the development of concurrent programs. However, these programs often fail because of concurrency bugs. A survey performed at Microsoft revealed that two-thirds of developers handle concurrency issues and bugs at least monthly [6]. Not only are concurrency bugs prevalent, but they often cause serious problems, such as Nasdaq's Facebook IPO glitch, which resulted in a loss of 40 million US dollars [1, 2].

Concurrency bugs are difficult to understand because they occur in non-deterministically executed code interleavings. In the Microsoft survey [6], more than 70% of developers rated reproducing concurrency bugs as a very difficult task. Even if developers reproduce a concurrency bug with a test, they often spend significant time understanding it (e.g., seven days on average in the Microsoft survey). Furthermore, developers often misunderstand a bug and create an incorrect patch. A recent survey on bug patches shows that patches for concurrency bugs are likely to be more buggy than patches for other types of bugs [26].

Some existing techniques locate likely concurrency bugs as memory-access orderings between multiple threads. Static techniques inspect memory accesses in concurrent programs that satisfy bug conditions [5,17]. Dynamic techniques monitor shared-memory accesses by executing concurrent programs and identifying likely buggy accesses [11, 16, 18, 19]. These techniques report concurrency bugs as interactions of memory accesses, such as pairs of memory accesses [11, 17], memory-access graphs [16], or memory-access patterns [5, 18, 19]. However, the techniques have several limitations. First, the techniques report only raw-memory accesses and lose context information, such as the call stacks of these accesses. Thus, developers must infer such information from the bug report to fully understand and fix the concurrency bugs. Second, the techniques do not identify memory accesses that are responsible for the same bug. Thus, a developer who receives a bug report with several memory accesses may need to manually filter false-positive accesses. Finally, the techniques do not explicitly handle multiple concurrency bugs and may miss reporting concurrency bugs when multiple bugs exist. Thus, developers may need to identify multiple bugs manually from the bug report.

Other existing techniques attempt to semi-automatically fix concurrency bugs [10, 12, 14]. The techniques input concurrency bugs reported from bug detectors [9, 27], select fix strategies, and produce patches that enforce correct orderings using synchronization operations. However, these techniques also have limitations. First, the techniques are not completely automatic, and require a developer's additional help. For example, developers may need to provide these techniques with a fix strategy after understanding the bug [12]. Second, the techniques use only simple strategies, such as lock-insertion. However, developers often use complex strategies, such as changing the design of data-structures related to the bugs [15]. Put another way, simple

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2483792

fixes may treat symptoms, but addressing the true cause of a bug may require more insight from the developer.

To address the limitations of existing techniques, we developed a new fault-comprehension technique, which we call Griffin. Griffin provides a way to explain concurrency bugs using additional information over memory accesses, and thus, bridges the gap between fault-localization and fault-fixing techniques. Griffin aids developers in understanding concurrency bugs by providing more information over fault-localization techniques, such as suspicious methods with which developers can easily locate atomic regions for atomicity violations. To assist automatic fault-fixing techniques, Griffin again provides more information, such as clusters of memory-access patterns, that can be used to generate only one patch for each cluster.

The key idea in Griffin is to group suspicious memory-access patterns. The process consists of three steps. In Step 1, Griffin executes an existing fault-localization technique to obtain suspicious memory-access patterns that represent concurrency bugs and a coverage matrix that shows the occurrences of the patterns in testing executions. In Step 2, Griffin uses the output of Step 1 to cluster executions that fail for the same concurrency bug. Finally, in Step 3, Griffin clusters patterns in the grouped failing executions (obtained in Step 2) and reports information for each likely bug in the program: suspicious memory-access patterns, suspicious methods from which suspicious memories are accessed, and bug graphs that show the interactions of groups of memory accesses.

There are three main advantages of Griffin over existing fault-localization techniques. First, Griffin *assists developers in understanding bugs* by providing summarized information for each bug. A developer begins to understand the bugs at a high level using a bug graph that shows context changes on groups of memory accesses. Next, the developer may investigate suspicious methods that are likely to contain the locations to be fixed. Finally, the developer may investigate each access pattern to understand each raw-memory access. Second, Griffin *handles multiple concurrency bugs* by grouping memory-access patterns. Thus, developers can concentrate on investigating patterns in a cluster, instead of investigating an unclustered list of patterns. Moreover, the identified different bugs might be assigned to several developers and handled concurrently [13]. Finally, Griffin *assists automatic fix techniques* by providing groups of fault-localization results. Thus, fix techniques can generate only one patch for each cluster instead of generating patches for each pattern and merging them for each bug afterward.

To evaluate Griffin, we implemented it for both Java and C++, and performed empirical studies on a set of real programs. The first study investigates how well Griffin clusters failing executions caused by the same bug. The results show that the technique accurately clusters failing executions with the parameters we used. The second study investigates how well Griffin clusters the suspicious patterns associated with the same bug, and presents the details of the bug. The results show that our technique presents suspicious patterns, suspicious methods, and bug graphs with only a few false positives. The third study investigates the efficiency of our technique in terms of the execution time. The results show that the technique completes the clustering and the explanation in a few minutes.

This paper makes the following contributions.

- It identifies problems with existing fault-localization technique for concurrent programs.
- It presents the Griffin technique, which clusters executions and patterns that are associated with the same bug, thereby providing a way to explain bugs.
- It carries out empirical studies that show the effectiveness and efficiency of the Griffin technique for clustering and explaining concurrency bugs.

2. BACKGROUND AND PROBLEM IDENTIFICATION

This section presents background for understanding our technique (Section 2.1), summarizes existing fault-localization techniques (Section 2.2), discusses three common problems with the existing techniques (Section 2.3), and discusses the challenges related to addressing these problems (Section 2.4).

2.1 Concurrency Bugs and Patterns

Our technique handles the two most important types of non-deadlock concurrency bugs: order violations and atomicity violations [15]. An *order violation* occurs when a pair of memory accesses between two threads, in which one of the memory accesses is a write, leads to unintended program behavior. An *atomicity violation* occurs when a set of accesses in an atomic region is interfered by other accesses in a different thread, and that interference leads to unintended program behavior. Researchers have identified *memory-access patterns* for both violations [24]. The complete list of patterns is given in Table 1 of Reference [19].

Figure 1 shows snippets from the `Vector` class of the Java Collection Library, which has an atomicity violation; statements in the code are labeled with their line numbers. Thread 1 executes the code on the left, and Thread 2 executes one of the three methods on the right. The constructor is the atomic region, and it must be executed without interference: `c` in lines 150–152 must be accessed without interference from the code in Thread 2. Otherwise, an atomicity violation can be triggered.

To illustrate, suppose the lines execute in order 150→271→151→851→852→152→681→682. In lines 150–151, the program retrieves `size` of `c` and initializes `array` of `Vector` with the `size`. However, in lines 851–852, the `size` and `array` of `c` are increased by the `addAll` method. Then, in lines 152–682, the increased `array` of `c` is copied to `array` of `Vector`, and thus, the program may crash with an exception. In this example, two patterns are manifested as atomicity violations. The first pattern involves a single variable, `size`: a read-write-read (RWR) pattern (i.e., the read accesses are from Thread 1, and the write access is from Thread 2), and the access order is 271→851→681. The second pattern involves two variables, `size` and `array`: a read-write-write-read (RWWR) pattern (i.e., the read accesses are from Thread 1, and the write accesses are from Thread 2), and the access order is 271→851→852→682.

2.2 Fault-Localization Techniques

This section briefly reviews four recent fault-localization techniques for concurrent programs, discussing how each presents its bug reports [11,16,19,22]. (Refer to the original papers for details.)

All four techniques collect memory accesses between threads

Thread 1

```
149  public Vector(Collection c) {
150    size = c.size();
151    array = new Object[size];
152    c.toArray(array);
153  }

270  public int size() {
271    return size;                    // read
272  }

680  public void toArray(Object a[]) {
681    a.size = size;                  // read
682    copyarray(a.array, array);      // read
683  }
```

Thread 2

```
850  public void addAll(Collection d) {
851    size += d.size;                 // write
852    appendarray(array, d.array);    // write
853  }

621  public void removeAllElements() {
622    size = 0;                       // write
623    array = null;                   // write
624  }

800  public void remove(int index) {
801    --size;                         // write
802    array[index] = null;            // write
803  }
```

Figure 1: Source code of `Vector`. Atomicity violations can occur when the execution of the atomic region in the `Vector` constructor (lines 150–152) is interfered by any of the three methods in Thread 2. For example, an atomicity violation can occur between `Vector` and `addAll` with the following order: 150→151→851→852→152.

Table 1: Memory-access patterns of fault-localization techniques.

Technique	Memory-access pattern	
	Pattern	Pattern accesses (additional information)
CCI [11]	R	681 (tag: thread-remote)
DefUse [22]	WR	851→681
Recon [16]	RWWRR	200→851→852→681→700 (most suspicious: 851→681)
Unicorn [19]	RWR	271→851→681

during program executions, and output a set of memory accesses ranked by suspiciousness. Table 1 shows the comparison of memory-access patterns of the four techniques for the example in Figure 1. The first column lists the techniques. The second column shows the memory-access patterns for each technique designated by the memory-access types (i.e., read (R) or write (W)). The third column shows the pattern accesses that are ranked 1st for each technique, along with additional information (in parentheses) if it is provided by the technique.

CCI [11] detects concurrency bugs using sampling and statistical methods. The technique samples memory-access locations, records each access along with a tag that indicates whether the previous access is thread-local or thread-remote, and records the execution output as passing or failing. Then, the technique computes the suspiciousness of memory-access locations using the statistics of execution output, and outputs a ranked list of the memory-access locations along with their associated tags. To illustrate, consider the first row in Table 1. CCI identifies the read access in line 681 of Figure 1 as the most suspicious location. CCI also reports a tag indicating that the read access in line 681 is thread-remote.

DefUse [22] detects concurrency bugs that violate definition-use (i.e., write and read, WR) invariants. The technique collects definition-use pairs between two threads in passing executions. Then, the technique finds the definition-use pairs in the failing executions that are not in the set of pairs in passing executions. To illustrate, consider the second row of Table 1. DefUse reports the bug in Figure 1 as the definition-use pair of `size` appearing in lines 851→681.

Recon [16] detects concurrency bugs using a form of a memory-access graph, called a context-aware communication graph. The graph shows five consecutive accesses of a memory location regardless of the memory-access type, and computes the most suspicious context change among the five accesses. Recon collects these memory accesses as graphs in multiple program executions, and ranks the graphs. To illustrate, consider the third row in Table 1. Recon reports these five consecutive accesses as a graph; for ease of presentation, we list them instead of showing them as a graph. Because Recon records all consecutive dynamic accesses near the bug without filtering non-crucial accesses to the bug, its output contains lines, such as 200 and 700, that do not appear in Figure 1. Recon also reports additional information: that 851→681 is the most suspicious thread-context edge among the five accesses.

Note that, to identify the atomicity violation, techniques should identify at least three memory locations: entry into an atomic region, interference, and exit out of the atomic region. In this example, the locations are accesses of `size` in 271→851→681. However, all three techniques mentioned above report only partial (i.e., one or two) memory-access locations that constitute the atomicity violation of `size`. In contrast, Unicorn reports *all* three accesses of `size` [19]. Unicorn detects order and atomicity violations using memory-access patterns. The technique monitors memory accesses with a fixed-sized window [18], and combines the accesses into patterns. Then, it computes the suspiciousness of the patterns, and ranks them. The fourth row of Table 1 shows the output of Unicorn. Unicorn reports all important memory-access locations of `size`.

2.3 Problems with Existing Techniques

The four fault-localization techniques of Section 2.2 share three common problems. Since Unicorn provides the most comprehensive reports, we illustrate these problems using Unicorn's reports, a sample of which appears in Table 2.

Table 2 shows the execution statistics of memory-access patterns and outputs reported by Unicorn for the example in Figure 1. The first column shows the pattern index (PI). The second to fourth columns are executions statistics. The second column reports the associated BugID (B), from 1 to 3, because there are three atomicity violations in the example. The third and fourth columns report the number of occurrences of the pattern in passing (P) and failing (F) executions, respectively. The fifth to ninth columns are

Table 2: Execution statistics for memory-access patterns and Unicorn output for the bugs in Figure 1.

PI	Statistics			Unicorn output			
	B	P	F	S	R	Pattern	Pattern accesses
1	1	0	3	0.60	1	RWR	271→851→681
2	1	0	3	0.60	1	RWWR	271→851→852→682
3	1	0	3	0.60	1	WR	851→681
4	1	0	3	0.60	1	RW	271→851
5	1	3	3	0.38	5	RW	852→682
6	2	0	2	0.20	6	RWR	271→622→681
7	3	0	2	0.20	6	RWR	271→801→681
8	2	0	2	0.20	6	RWWR	271→622→623→682
9	3	0	2	0.20	6	RWWR	271→801→802→682

outputs of Unicorn. The fifth and sixth columns show the suspiciousness score of the pattern (S) and the ranking of the pattern (R), respectively. The seventh column reports the type of the pattern, with memory access types. The final column reports the access locations of the pattern in the source code.

Problem 1 (Loss of Context): The first problem is that existing techniques report memory-access locations but not the context of the bug. Section 2.2 explains that Unicorn outputs *all* raw memory-access locations that are likely associated with concurrency bugs, but even all raw memory-access locations may not fully explain the cause of the bug. To illustrate, consider again the example in Figure 1. To understand the bug, the developer would inspect the code using one of the four first-ranked patterns in Table 2.[1] Then, the developer would locate the `size` and `toArray` methods, but may not easily find the constructor using these accesses. Figure 2 shows the same memory accesses of `size` along with the call stack for each memory access. Solid boxes represent methods on the call stack, and dashed boxes represent memory accesses. Without inspecting the figure, developers may not find that `Vector` is the common method that leads to the two raw-memory accesses.

Problem 2 (True-/False-positive Patterns): The second problem is that existing techniques report the ranked list of the patterns, but do not group patterns responsible for the bug. Consider again the example in Figure 1. As discussed in Section 2.1, two patterns can manifest the same atomicity violation. PIs 1 and 2 are atomicity violation patterns involving a single variable (i.e., `size`) and multiple variables (i.e., `size` and `array`), respectively. PIs 3–5 manifest the same violation, and the pattern accesses are part of PIs 1 and 2. Thus, PIs 1–5 manifest as one bug: B1. However, neither Unicorn nor the other techniques [11, 16, 22] reveal that these patterns are associated with the same bug.

Problem 3 (Multiple Bugs): The third problem deals with multiple bugs. PI6 to PI9 in Table 2 illustrate the problem. The PIs are associated with two different bugs. PI6 and PI8 are associated with B2, and PI7 and PI9 are associated with B3. B2 is an atomicity violation that occurs between the `Vector` and the `removeAllElements` methods. B2 occurs when an execution order is 150→`removeAllElements`→152. Similarly, B3 is an atomicity violation that occurs between the `Vector` and the `remove` methods. B3 occurs with an execution order, 150→`remove`→ 152. When multiple bugs exist, the patterns manifesting different bugs can be listed with the same or similar suspiciousness score, and developers may need to associate the patterns with different bugs manually.

[1]Like many fault-localization techniques, if patterns result in the same suspiciousness, Unicorn gives them the same rank, and thus, they would be inspected in any order.

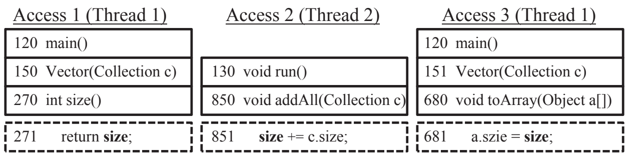

Figure 2: Problematic memory accesses with call stacks.

2.4 Challenges

The problems of Section 2.3 raise a number of challenges.

Challenge 1 (Efficient Information Gathering): Existing techniques collect and report only pairs of memory accesses or patterns of accesses (in Unicorn). This property helps to keep the amount of information gathered small, making the techniques efficient overall. However, to present concurrency bugs with high-level context and clusters of accesses, a new technique needs to collect more context information with accesses efficiently and process them accordingly.

Challenge 2 (Large Context): The second challenge is related to large size of calling contexts. It may be argued that Problem 1 can be easily addressed by reporting only a call stack for each memory access. However, the problem is not so easy. The stack sizes sometimes grow very large, such as `mysql-169`, where the size of the call stacks grow to over 10 methods spread across five source files. In addition, the stack comparison involves not only stacks of two memory accesses, but stacks of patterns of memory-accesses, and moreover, the number of stacks grows to several thousands.

Challenge 3 (Large Number of Patterns): The third challenge is related to the large number of patterns. It may be argued that Problems 2 and 3 can be easily addressed by manual clustering of patterns by a developer. However, it may be difficult to group patterns manually when the size of the report grows. Our experience suggests that the size of a report, like `mysql-791` in Table 3, can grow to more than 10K in some subjects [19], and the size of the patterns responsible for the same bug may grow accordingly.

3. OUR TECHNIQUE

This section first presents an overview of our technique, Griffin, that clusters memory-access patterns and outputs these patterns along with suspicious methods and a bug graph for each concurrency bug (Section 3.1). Then, it presents the details of the three steps of the technique (Sections 3.2–3.4).

3.1 Overview

We designed Griffin to address the three problems with existing techniques (see Section 2.3). Griffin addresses Problem 1 by providing high-level context over raw-memory accesses: groups of suspicious patterns, suspicious methods,

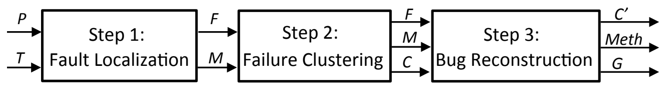

Figure 3: Overview of Griffin.

and bug graphs. Griffin addresses Problems 2 and 3 with two clustering algorithms that group memory-access patterns that fail for the same concurrency bug.

Griffin consists of three steps as shown in Figure 3. Step 1 takes as input a concurrent program P and a test suite T. Griffin executes P with T multiple times, collecting memory-access patterns from each execution and labeling executions as passing or failing. Then, Griffin computes a ranked list of patterns, F, using a fault-localization scheme, and generates a coverage matrix, M, that represents the associations of the patterns and the executions. Step 2 uses F and M to cluster executions that likely failed due to the same bug. This clustering is based on Algorithm 1, ClusterFailingExecutions (Algorithm 1), and underlies Griffin's ability to handle multiple bugs. The output of Step 2 is a set of clusters, C, where each cluster contains a set of failing executions. Finally, Step 3 executes algorithm ClusterSuspiciousPatterns (Algorithm 2) on F, M, and C. This algorithm groups patterns for constructing higher-level context that will be of direct use to the developer in identifying the bug. The output C' is a set of clusters, where each cluster contains a set of patterns. For each cluster in C', Step 3 also reports two suspicious methods (one for each thread in the cluster), $Meth$, that are likely to contain the location to be fixed, and a bug graph G, that visually shows the bug.

3.2 Step 1: Localize Problematic Patterns

Step 1 is based on the Unicorn [19] fault-localization technique. We extended Unicorn in two ways to provide F and M. First, we extended Unicorn to record call-stack information for each shared-memory access, and thus, accesses in the patterns in F are also associated with call stacks. This modification is straightforward and is used in other techniques [10,12,14].

Second, we extended Unicorn to report M. Unicorn maintains M internally in its pattern-combination step, so we changed Unicorn to output the information. Figure 4 shows an example of M, for the bug example in Figure 1 and Table 2. In the figure, the rows represent the pattern indexes (PI1–PI15); the first 15 columns represent the executions (E1–E15); the labels at the bottom of these columns represent the results of those executions (P for passing and F for failing); and the last two columns represent the suspiciousness score and the rank of the PIs, respectively. A solid circle in a cell in M, denoted by (PI,E), indicates that PI is observed in the execution of E. For example, the solid circle in (PI1, E1) indicates that pattern accesses of PI1 were observed in the execution E1, the suspiciousness score of PI1 is 0.6, and PI1 is ranked 1st. Note that Figure 4 has nine PIs in common (i.e., PI1–PI9) with Table 2, but it has six more PIs (i.e., PI10–PI15) for illustration of our technique.

3.3 Step 2: Cluster Failing Executions

The main purpose of Step 2 is to handle multiple bugs by grouping executions that fail for the same reason. To do this,

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	Susp	Rank
PI1	●	●	●													0.60	1
PI2	●	●	●													0.60	1
PI3	●	●	●													0.60	1
PI4	●	●	●													0.60	1
PI5	●	●	●					●	●			●				0.38	5
PI6				●	●											0.20	6
PI7						●	●									0.20	6
PI8						●	●									0.20	6
PI9				●	●											0.20	6
PI10				●	●			●	●			●	●	●	●	0.15	10
PI11						●	●		●	●	●	●	●	●		0.15	10
PI12				●		●		●	●	●	●	●	●		●	0.14	12
PI13					●		●	●	●		●	●	●	●	●	0.14	12
PI14				●			●	●	●	●	●	●	●	●	●	0.13	14
PI15	●							●	●	●	●	●	●	●	●	0.06	15
	F	F	F	F	F	F	F	P	P	P	P	P	P	P	P		

Figure 4: Coverage matrix for the atomicity violations in Figure 1 and Table 2.

Step 2 executes a clustering algorithm based on the results of Step 1, and outputs a set of clusters of failing executions for use in Step 3.

There are several metrics to use for clustering algorithms, such as statement, branch, definition-use profiles, memory-access pairs between threads, and memory-access patterns. We tried different metrics and used the fault-localization results (i.e., ranked memory-access patterns) to develop our clustering algorithm for two reasons. First, fault-localization results are available from Step 1, so Step 2 can use them without collecting additional profile data. Second, concurrency bugs usually occur in specific thread interleavings and are related to a small portion of program profiles, but are not highly related to the entire profile of the program execution.

Figure 4 illustrates the intuition behind our approach. Executions E1–E7 are failing executions: E1–E3 fail because of the same bug; E4 and E5 fail because of the same bug; and E6 and E7 fail because of the same bug. Each failing execution associated with the same bug has a similar set of top ranked patterns. Consider E1–E3. PI1–PI4 are highly ranked and appear only in these executions. In contrast, PI10–PI15 are lowly ranked, and mainly appear in passing executions. Even if the pattern accesses appear in failing executions, they appear randomly. Therefore, using the top ranked patterns for each failing execution will facilitate the clustering of executions that fail for the same bug.

Algorithm 1, our fault-localization-based clustering algorithm, inputs a ranked list of memory-access patterns, F, a coverage matrix, M, the number of suspicious patterns to be considered, t, and the threshold of similarity of two clusters, s, and outputs a set of clusters, C=$\{C_1, C_2, \ldots, C_k\}$, where each cluster C_i is a set of failing executions. The algorithm starts by initializing each cluster to contain one failing execution (lines 1–4). Then, the algorithm iterates until there are no more merged clusters (lines 5–27). In this main loop, the algorithm compares every pair of clusters to determine whether they can be merged (lines 10–26). To check the similarity, the algorithm fetches the top t patterns in two clusters (lines 11–12), and computes the similarity of the two clusters (line 13). For the similarity computa-

Algorithm 1: ClusterFailingExecutions

```
Input  : $F$: ranked list of memory-access patterns
         $M$: coverage matrix
         $t$: number of suspicious patterns to be considered
         $s$: threshold of similarity of two clusters
Output: $C$: set of clusters, each of which is a set of failing
        executions
Data:  $k$: number of clusters
       $merged$: true if any two clusters were merged
1  Set $k$ = number of failing executions
2  for failing executions, $f_i$, where $1 \leq i \leq k$ do
3      $C_i = \{f_i\}$
4  end
5  $merged$ = true
6  while $merged$ do
7      $merged$ = false
8      $max\_similarity = 0$
9      $cand\_pair = \emptyset$
10     for all ($C_i$, $C_j$) where $i \neq j$ and $0 \leq i, j \leq k$ do
11         $R_i$ = getSuspiciousPatterns($C_i$, $M$, $F$, $t$)
12         $R_j$ = getSuspiciousPatterns($C_j$, $M$, $F$, $t$)
13         $similarity$ = getSimilarity($R_i$, $R_j$)
14         if $similarity > max\_similarity$ then
15             $cand\_pair = (C_i, C_j)$
16             $max\_similarity = similarity$
17         end
18         if $max\_similarity == 1$ then
19             break
20         end
21     end
22     if $max\_similarity \geq s$ then
23         merge $cand\_pair$
24         $k = k$ - 1
25         $merged$ = true
26     end
27 end
28 C = $\{C_1, C_2, \ldots, C_k\}$
29 return C
```

Algorithm 2: ClusterSuspiciousPatterns

```
Input  : $F$: ranked list of memory-access patterns
         $M$: coverage matrix
         $C_f$: a set of failing executions
         $t$: number of suspicious patterns to be considered
Output: $C'$: set of clusters, each of which is a set of
        patterns
Data:  $k$: number of clusters
1  Set $k = t$
2  $C'$ = getSuspiciousPatterns($C_f$, $M$, $F$, $t$)
3  for patterns $p_i$ in $C'$ do
4      $CL_i = \{p_i\}$
5  end
6  $merged$ = true
7  while $merged$ do
8      $mergeable$ = false
9      for all ($CL_i$, $CL_j$) where $i \neq j$ and $0 \leq i, j \leq k$ do
10         $mergeable$ = isMergeable($CL_i$, $CL_j$)
11         if $mergeable$ then
12             merge($CL_i$, $CL_j$)
13             $k = k$ - 1
14             $merged$ = true
15             break
16         end
17     end
18 end
19 $C' = \{CL_1, CL_2, \ldots, CL_k\}$
20 return $C'$
```

tion, the algorithm uses the Jaccard similarity index, which is shown in Equation 1. The value of the index ranges from 0 (completely different) to 1 (completely alike).

$$\text{Similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|} \qquad (1)$$

The algorithm keeps the maximum similarity by comparing the similarity of the current pair and *max_similarity* (lines 14–17). If the two clusters are exactly the same (lines 18–20), the algorithm simply merges the two clusters (lines 22–26). Otherwise, if the maximum similarity is greater than the threshold (lines 22–26), the algorithm merges the two clusters. The algorithm terminates the main loop when no more clusters are merged, and returns the clusters (line 28).

Consider again Figure 4. Algorithm ClusterFailingExecutions starts by initializing each cluster to one failing execution: for $i \in 1 \ldots 7$, C_i={Ei}. Suppose t=5 and s=0.5. Consider C_1 and C_2 as the first pair of clusters. These clusters can be merged because the five top-ranked patterns of each cluster are exactly the same and thus, the similarity is 1 (lines 18–20 and 22–26). Let C'_1={E1, E2}. Note that the algorithm breaks out of the comparison loop if it finds two identical clusters (line 15). Now consider C'_1 and C_3 as the next pair of clusters. This pair can also be merged because the five top-ranked patterns of each cluster are exactly the same resulting in a similarity of 1. Let C''_1={E1, E2, E3}. Another candidate pair is C_4 and C_5. E4 has five patterns, and E5 has four patterns. Because three patterns appear in both E4 and E5 and six distinct patterns appear in E4 and E5, the similarity is $\frac{3}{6}$ or 0.5. Thus, the pair can be merged. Likewise, E6 and E7 can be merged. Finally, the algorithm reports three clusters: C''_1={E1, E2, E3}, C'_2={E4, E5}, and C'_3={E6, E7}.

3.4 Step 3: Reconstruct Bug Context

The main purpose of Step 3 is to reconstruct high-level bug context from the clustered patterns to assist understanding of the bug. To do so, Step 3 fetches the top-ranked patterns for each group of failing executions reported from Step 2, and distinguishes the true and false positive patterns with respect to the bug. Step 3 outputs a set of clusters of patterns, C', for each cluster in C. Then, for each cluster of patterns in C', Step 3 merges accesses of the patterns, and reports suspicious methods, $Meth$, and a bug graph, G.

We use a clustering algorithm based on the similarity of the call stacks [4]. The intuition behind the call-stack-similarity-based clustering is that accesses appearing closely in execution are likely to have similar call stacks, and thus, accesses responsible for the same concurrency bug are likely to have similar call stacks. The same intuition is applied to patterns. If two patterns are responsible for the same bug, the call stacks in the patterns are likely to be similar. Specifically, we use common call stacks to compare patterns. A *common call stack* of a group of memory accesses is the common part of the call stacks from the bottom of the stacks. Consider accesses 1 and 3 in Figure 2. The common call stack of the two accesses is the common part of the two call stacks of the accesses: `main()` and `Vector()`.

Algorithm 2, our suspicious-pattern clustering algorithm, inputs a ranked list of memory-access patterns F, a coverage matrix M, the number of suspicious patterns to be considered t, and a set of failing executions, C_f, that fail because of the same bug. The algorithm starts by fetching the top t patterns from C_f and initializing each cluster to a pattern (lines 1–4). Then, the algorithm iterates until there are no more merged clusters (lines 7–17). In the main loop, the algorithm compares every pair of clusters to determine whether the pair can be merged (lines 9–15). The algorithm terminates the main loop when no more clusters are merged, and returns the clusters (line 19).

The main part of the algorithm is the `isMergeable` function that determines whether two clusters of patterns can be merged. Consider two clusters, CL_i and CL_j. There are two cases. First, if CL_i and CL_j have common call stacks that are the same, the clusters can be merged. Second, if CL_i has only one pair and the pair in CL_i is part of the triple or quadruple in CL_j, then the clusters can be merged.

Consider again Table 2 and Figures 2 and 4 that were used in Step 2. Step 3 inputs C_1''={E1, E2, E3}, C_2'={E4, E5}, and C_3'={E6, E7}. Suppose t=5, and we run Algorithm 2 for C_1''. The top five patterns in C_1'' are PI1–PI5. The algorithm starts by initializing each cluster with each pattern: for $i \in 1 \ldots 5$, CL_i={PIi}. First, the algorithm compares CL_1 and CL_3. Each cluster has one pattern, and PI3 (i.e., 851→681) is part of PI1 (i.e., 271→851→681). Thus, they are merged by exact subsumption, and let CL_1'={PI1, PI3}. Similarly, PI4 and PI5 are merged to PI2, and let CL_2'={PI2, PI4, PI5}. Now, the algorithm compares CL_1' and CL_2', and merges them by call stack similarity. The common call stacks of the clusters are: `main()` and `Vector()` from one thread; `run()` and `addAll()` from the other thread as in Figure 5. Finally, the algorithm reports only one cluster, CL_1''={PI1, PI2, PI3, PI4, PI5}, for C_1''.

Step 3 reports two suspicious methods *Meth* one for each thread in the cluster. We define a *suspicious method* as the method at the top in the common call stack. A cluster has two groups of memory accesses for each thread, and thus, Step 3 finds two suspicious methods, one for each thread. Consider again Figure 2. Suppose a cluster has only these three accesses. Then, a group of memory accesses for Thread 1 is Access 1 and Access 3, the other group of memory accesses for Thread 2 is Access 2, and the suspicious methods are `Vector()` and `addAll()`.

Step 3 reports a *bug graph G* for each cluster of patterns. To create the graph, Step 3 divides memory accesses into two groups, one for each thread. Then, Step 3 investigates memory-access orderings and groups accesses that have the same ordering sequences. Figure 5 shows an example of a bug graph for the cluster, CL_1''. Each node has a group of memory accesses with a common call stack. Solid rectangles represent methods on the call stack, and dotted rectangles represent memory accesses. For example, the node in Thread 2 has two write accesses, lines 851 and 852, and their common call stack has two methods. Edges between threads show the orderings of the memory accesses, and in this example, the edges can represent all five patterns in PI1–PI5. For example, PI1 is a RWR pattern of 271→851→681 accesses, and we can find the corresponding edges of the arrows in the graph.

4. EMPIRICAL STUDY

To evaluate the GRIFFIN technique, we implemented it in a prototype tool, and evaluated the tool on a set of subjects.[2]

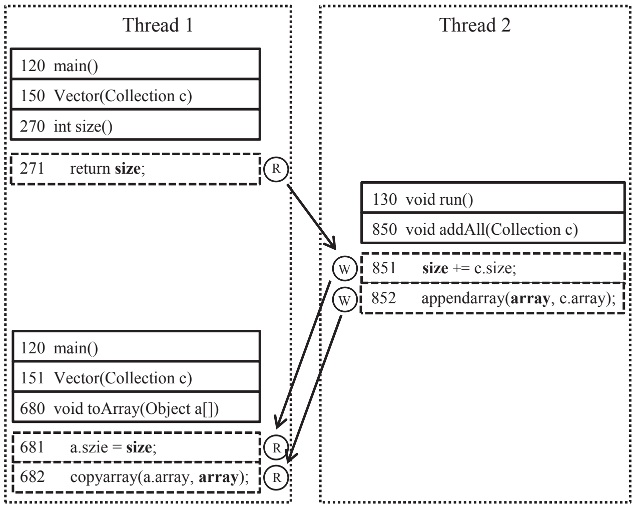

Figure 5: Bug graph, G, for the atomicity violation B1 in Table 2.

This section describes the implementation (Section 4.1) and the empirical set up (Section 4.2), presents our studies (Sections 4.3–4.5), and discusses the threats to the validity of the studies (Section 4.6).

4.1 Implementation

We implemented our prototype tool for Java and C++. For Step 1 of our technique, we implemented modules for both Java and C++. The modules are based on the implementations of UNICORN [19], and we extended them to record call stacks for each shared-variable access and to provide the coverage matrix. For both languages, Step 1 outputs the results in XML format. We implemented the module using the Soot framework[3] and the PIN binary instrumentation tool[4] for Java and C++, respectively.

For Step 2, we implemented the module in Java. The module takes XML files containing the results of Step 1 as input, and produces as ouput a text file of clusters of failing executions. Finally, for Step 3, we implemented the module in Python and GraphViz. The module reads the XML and the text files, and produces clusters of patterns and suspicious methods in text files, and bug graphs in PNG graphics files.

4.2 Empirical Setup

Table 3 lists the subject programs we used for our studies. The first column shows the subject's language (Language), either Java or C++. The second column shows the name of the subject program (Program). The third column shows the failure rate that we observed empirically with our test cases (%F). The fourth column shows the number of concurrency bugs that we identified with our test cases (#B). The fifth column lists the size of the program in KLOCs. Finally, the sixth column shows the type of the concurrency bug (T), either an atomicity (A) or an order (O) violation. For example, `TreeSet-1` is a Java program that fails 42% of the time with our test suite. The size of the program

[2]The details of our empirical studies are available at `http://www.cc.gatech.edu/~sangminp/issta2013/`

[3]`http://www.sable.mcgill.ca/soot/`

[4]`http://pintool.org/`

Table 3: Subjects used in the empirical evaluation.

Language	Program	%F	#B	KLOC	T
Java	TreeSet-1	42.0	5	7.5	A
	TreeSet-2	29.0	3	7.5	A
	StringBuffer-1	33.0	4	1.4	A
	StringBuffer-2	18.0	1	1.4	A
	Vector-1	8.0	4	9.5	A
	Vector-2	14.0	2	9.5	A
C++	Mysql-169	29.0	1	331	A
	Mysql-791	24.0	1	372	A
	NSPR-165586	18.0	1	125	A
	PBZip2	75.0	1	2	O
	Transmission	31.0	1	90	O

Table 4: Study 1 (Handling Multiple Bugs).

Program	#Patterns	$\#O$	$\#C$	F-measure
TreeSet-1	714	5	7	0.88
TreeSet-2	656	3	4	0.91
StringBuffer-1	12	4	4	1.00
StringBuffer-2	3	1	1	1.00
Vector-1	18	4	4	1.00
Vector-2	10	2	2	1.00
Mysql-169	21834	1	1	1.00
Mysql-791	71694	1	2	0.94
NSPR-165586	1479	1	2	0.86
PBZip2	427	1	2	0.96
Transmission	226	1	1	1.00

is about 7,500 lines of code, and it fails with five different atomicity violations.

We created test cases, and ran them multiple times for each subject to collect various interleavings from multiple executions. For the Java programs, which are a set of classes in the Java collection library, we created two test cases for each program. These test cases execute different parts of code with different levels of concurrency, and thus, programs with different test cases may exhibit different sets of bugs. Specifically, the program name with suffix 1 has a test case that creates four threads and calls two methods for each thread, and the program name with suffix 2 has a test case that creates two threads and calls one method for each thread. `Mysql-169` and `Mysql-791` are two versions of `Mysql` open-source database, and `NSPR-165586` is a library of the Mozilla open-source browser. For each program, we created test cases as described in the program's bug repository to emulate the real debugging environment of the developers. `PBZip2` is a compression utility program, and we provided a large file for the program to compress. `Transmission` is a torrent download utility, and we used a torrent file for the program to download and observed whether the program fails.

We ran our studies on a Linux desktop with 2.8 GHz CPU and 8 GB of memory. We used gcc 4.1 and Java 1.5.

4.3 Study 1: Handling Multiple Bugs

The goal of this study is to investigate how well GRIFFIN handles multiple concurrency bugs by examining the clusters of failing executions it produces. More specifically, we collected the fault-localization results from Step 1 of our technique, and ran Step 2 on the results. Recall that Algorithm 1 in Step 2 has two parameters, t and s: t is the number of suspicious patterns to be considered for each failing execution, and s is the threshold of similarity of two clusters. We performed a preliminary study that investigates a range of values of t and s to determine the parameter values that are likely to report the same number of clusters as the number of bugs. From this study, we let $t = 30$ and $s = 0.8$.

To evaluate the effectiveness of the clustering, we used the F-measure metric. This metric is based on a combination of precision and recall as used in the information-retrieval domain, and is widely used to evaluate clustering algorithms [4,23]. In this case, *precision* refers to the proportion of clustered failing executions that are relevant, *recall* refers to the proportion of relevant failing executions that are clustered; the *F-measure* is a weighted combination of precision and recall.

We denote C as the set of clusters. Specifically, let C_i be the ith cluster that our algorithm reports, O_j be the jth optimal cluster, and N be the total number of failing executions. Then, we calculate precision and recall as follows.

$$\text{Precision}(C_i, O_j) = \frac{|C_i \cap O_j|}{|C_i|},\ \text{Recall}(C_i, O_j) = \frac{|C_i \cap O_j|}{|O_j|}$$

We use Van Rijisbergen's F-measure [23], which computes the weighted average of maximal F-measure values for each cluster. The equations are as follows.

$$F(C_i, O_j) = \frac{2 * \text{Recall}(C_i, O_j) * \text{Precision}(C_i, O_j)}{\text{Recall}(C_i, O_j) + \text{Precision}(C_i, O_j)}$$

$$F\text{-measure}(C) = \sum_i \frac{|C_i|}{N} * \max_j\{F(C_i, O_j)\}$$

The F-measure value ranges from 0 (least effective) to 1 (most effective).

Table 4 shows the results of the study. The first column shows the program name (Program). The second column shows the number of patterns (#Patterns) that the fault-localization technique in Step 1 reports. The third column shows the optimal number of failing execution clusters ($\#O$), which should be the same as the number of bugs in the program. The fourth column shows the number of clusters that GRIFFIN reports ($\#C$). Finally, the fifth column shows the F-measure value (F-measure). For example, for `Mysql-791`, GRIFFIN reports 71,694 patterns after Step 1. The optimal number of clusters is one, and Step 2 of the technique reports two clusters. The F-measure value is 0.94.

The main observation from Table 4 is that most of the F-measure values are 1.00 or are close to 1.00, and none is less than 0.86. The high F-measure values indicate that the failure clustering algorithm is effective in handling multiple concurrency bugs with our parameter settings. One important parameter is t, which we set to 30. Although previous techniques [10,12,16,18,19] reveal that several patterns may indicate the same bug, they do not report the number of patterns that are related to the same bug. Our study showed that, for our subjects, up to 30 patterns are related to the same bug.

We performed a detailed investigation of the subjects whose F-measure values are not 1.00. For example, for `Mysql-791`, we used 24 failing executions as input to the clustering algorithm. The optimal number of clusters is one, but the algorithm reported two clusters, one with 23 failing executions and the other with one failing execution. We manually investigated these clusters, and found that these two clusters

Table 5: Study 2 (Reconstructing Bug Context).

Program	$\#O$	$\#C$	FP	Meth	Call Size
TreeSet-1	5	5	0	Y	6
TreeSet-2	3	3	0	Y	6
StringBuffer-1	4	4	0	Y	1
StringBuffer-2	1	1	0	Y	1
Vector-1	4	4	0	Y	1
Vector-2	2	2	0	Y	1
Mysql-169	1	2	1	Y	9
Mysql-791	1	1	0	Y	1
NSPR-165586	1	1	0	Y	4
PBZip2	1	1	0	Y	0
Transmission	1	1	0	Y	7

are also similar. We also found that, when we used a lesser value for s (the threshold of similarity), we get the optimal number of clusters for the subject. Thus, we believe that there may be a better combination of parameters that more often gives optimal results, and we plan to investigate this in future work.

Another observation involves the "super-bug effect" [28]: for sequential programs, the clustering techniques often did not cluster failing test cases effectively when the program has multiple bugs, and one bug hides the appearance of the other bug. For concurrent programs and test cases we investigated, we found no executions that reveals a super bug.

4.4 Study 2: Reconstructing Bug Context

The goal of this study is to investigate how well GRIFFIN reconstructs bug contexts. Specifically, this study investigates how well GRIFFIN clusters true positive patterns and locates concurrency bugs in both method and access levels. To do so, we run Step 3 of the technique on the optimal clusters of failing executions that Step 2 produces. Recall that Algorithm 2 in Step 3 has a parameter t, which is the number of patterns to be re-clustered. We set $t = 20$ based on the observation that only top-ranked patterns are likely to be responsible for the real bugs.

To evaluate the effectiveness of the technique, we investigated the outputs of Step 3: clustered patterns, suspicious methods, and bug graphs, using our understanding of the bugs. To evaluate the effectiveness of the clustering, we checked whether the bug graph of the cluster has orderings and contexts consisting of the bug. If the bug graph does not represent a bug, we set it as a false positive. To evaluate the effectiveness of the bug locating ability, we manually checked whether the suspiciousness method locates the method containing the cause of the bug. For example, if the bug is an atomicity violation, we checked whether the suspicious method locates the atomic region; if the bug is an order violation, we checked whether the suspicious method has the location in which incorrect orderings occur.

Table 5 shows the results of the study. The first column shows the program name (Program). The second column shows the number of optimal clusters of patterns ($\#O$), which is the same as the number of bugs. The third column shows the number of clusters that GRIFFIN reports ($\#C$), The fourth column shows the number of false-positive clusters (FP), which is the difference between the values in the third and second columns. The fifth column shows whether the suspicious methods locate the cause of the bug (Meth). Finally, the sixth column shows the maximum size of call stacks between suspicious methods and raw-memory accesses (Call Size). For example, GRIFFIN reports two clusters for `Mysql-169`, where the patterns in one cluster indicate the real bug, and the other cluster is a false positive. The suspicious methods locate the atomic region, and the maximum size of call stacks between suspicious methods and raw-memory accesses is 9.

We make several observations from the study. First, for our subjects, the technique successfully clusters patterns. The clusters included *all* 24 of the true bugs across all subjects, with just 1 false positive cluster. Thus, for each bug, a developer would typically need only to investigate one bug graph to understand the bug in a high level, and investigate only up to 20 patterns to understand the bug in a low level. However, without such clustering, a developer might need to investigate an indefinite number of patterns—up to 71k patterns as shown for `Mysql-791` in Table 4.

For the subjects we investigated, we found only one false positive in `Mysql-169`. The bug is a multi-variable atomicity violation.[5] Our manual investigation found that the bug graph for the true-positive cluster represents the real bug as given in the bug description. However, the bug graph for the false-positive cluster shows two thread-context changes, where the context changes resemble those of the real bug but differ slightly in that the accesses in the false-positive cluster were observed closely in time. (The details of this bug are given at the link in Footnote 3.)

Second, for our subjects, we found that all suspicious methods locate the method that is the cause of bug. For `Vector-1` and `Vector-2`, suspicious methods include the `Vector` constructor. For another example, for `Transmission`, GRIFFIN reports `tr_sessionInitFull` as the suspicious method, where the incorrect ordering actually occurs.[6]

Third, for our subjects, call-stack sizes between suspicious methods and raw-memory accesses are greater than zero except for `PBZip2`. Considering the large call stack sizes in Table 5 and the large number of patterns in Table 4, we can confirm that Challenges 2 and 3 in Section 2.4 are not trivial.

4.5 Study 3: Efficiency

The goal of this study is to investigate the efficiency of GRIFFIN in terms of time overhead. Specifically, we measured the execution time for Steps 2 and 3. We did not measure the time for Step 1 because we used the UNICORN implementation, which takes 2× and 14× overhead for Java and C++ programs, respectively. The time overhead for Step 1 is comparable to that of existing techniques [16, 18].

The results of the study show that Steps 2 and 3 are each performed in less than one minute. The maximum time taken for Step 2 is for `Mysql-791`, which took only three minutes. The maximum time for Step 3 is less than one minute. These results indicate that, for our subjects, our technique is efficient in that it adds less than five minutes to the time of fault-localization techniques.

4.6 Threats to Validity

There are several threats to validity of our studies. Threats to internal validity include the empirical setup, especially the test suite that we used for our studies. Because there

[5] `http://web.eecs.umich.edu/~jieyu/bugs/mysql-169.html`

[6] `http://web.eecs.umich.edu/~jieyu/bugs/transmission-142.html`

is no standard benchmark suite for use in fault-localization of concurrency bugs, we used programs with known bugs. However, to mitigate this threat, we created and used the test suites as described in the bug description of the repositories to apply our technique to test suites of open-source programs.

Threats to external validity limit the generalization of the technique in real debugging situations. Because of the lack of a standard benchmark suite, this problem is common to all existing work in the area. However, to reduce this threat, we chose programs with real bugs, and we used programs for both Java and C++.

Threats to construct validity include the way developers utilize the outputs of the fault-localization techniques. Our technique provides three different forms of output, so developers may understand bugs in different views with our results. In addition, we evaluated our technique with a set of experiments, showing that our technique pinpoints bugs in both method and memory-access levels. However, we acknowledge that only user studies will inform us about the usefulness and effectiveness of our techniques.

5. RELATED WORK

This section discusses existing fault-localization and fault-fixing techniques, along with fault-clustering and fault-comprehension techniques, and compares them with Griffin.

Fault localization. Early work on fault-localization techniques for concurrent programs located one type of concurrency faults. These techniques statically inspect program code [5, 17] or dynamically monitor shared-memory accesses [7, 25], and report a list of problematic memory-accesses patterns representing data races [17] or atomicity violations [5,7,25]. More recent work (see Section 2.2) investigates memory-access patterns for several types of concurrency bugs and outputs these patterns as bug reports [11,16, 19,22]. All these fault-localization techniques report a list of memory-access patterns as bugs. However, Griffin clusters these patterns for each bug and provides further contextual information to improve comprehension of concurrency bugs.

Automatic fix. There are a few recent techniques for automatic fixing of concurrent programs [10,12,14]. AFix [10] and AXis [14] input atomicity violations from the output reports of existing techniques [9], determine the atomic regions from the reports, and insert locks to protect atomic regions. CFix [12] extends AFix to handle both atomicity and order violations. Griffin differs from these techniques in two ways. First, the goal of Griffin is to enhance understanding of bugs by providing clusters of patterns and suspicious method calls, and to let developers fix the bug with various strategies. In contrast, these fix techniques assume complete understanding of bugs, and use simple strategies such as lock-insertion, that developers rarely use [15]. Second, Griffin can help automatic fix techniques because it provides a more complete explanation of bugs, which can be input to these techniques.

Fault clustering. One of the main parts of Griffin is clustering failing executions, and there are largely two types of techniques for the clustering as discussed in Section 3.3: profile-based clustering and fault-localization-based clustering. Podgurski and colleagues [21] showed that program profiles can be used for clustering failing executions according to the causes of the faults. Zheng and colleagues [28] presented a profile-based clustering algorithm that finds bug predicates and that handles multiple bugs. Jones and colleagues [13] utilized both profile-based and fault-localization-based clustering to improve both sequential and parallel debugging, and showed that both algorithms save debugging cost over non-clustered techniques. Our technique uses a fault-localization-based clustering technique because fault-localization data is available from an existing technique. However, unlike these techniques, our technique also uses a pattern clustering algorithm to further refine the clustering results.

Fault comprehension. There are several approaches that improve understanding of bugs by providing additional information over fault-localization results for sequential programs. The underlying idea of the approaches is that providing suspicious statements is not enough to identify and understand the bug, and that providing additional information (e.g., clustering results and explanations) can improve developers' understanding [20]. Rapid [8] provides suspicious statements with the method context leading to the statement so that developers can infer the execution path to the bug. LEAP [3] provides suspicious graphs at the method and the basic-block levels using graph-mining algorithms so that developers can understand the bug at a level higher than statements. Like these approaches, Griffin provides additional information over fault-localization techniques. However, unlike these techniques, Griffin works for concurrent programs.

6. CONCLUSION AND FUTURE WORK

In our view, Griffin makes significant progress toward filling an important research gap in *fault-comprehension*, which lies between fault-localization and fault-fixing techniques for concurrent programs. Recall that, to date, fault-localization has focused on pinpointing basic program fragments (memory references) that lead to bugs, and fault-fixing techniques attempt (semi-)automatic repair through the use of simple patches. However, experience suggests that true fixes require deeper program understanding, which Griffin attempts to provide through explicit identification of suspicious *methods* rather than just memory references; reconstruction of *bug context* through analysis of call stacks, rather than just reporting call stacks; and the use of bug graphs to aid *developer* understanding of this information.

There are several directions for future work that can improve Griffin. First, we used several subject programs with multiple bugs, but the programs may not be sufficient to show various properties of rankings of fault-localization techniques. Thus, we plan to investigate more real-world subjects that involve multiple bugs. Second, we analytically showed that our results facilitate understanding of concurrency bugs by providing more summarized information than existing fault-localization techniques. However, we plan to perform empirical user studies to determine how well our results assist developers' in understanding and fixing concurrent bugs.

Acknowledgements

We thank Fuxin Li for pointing us to F-measures. 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.

7. REFERENCES

[1] Nasdaq outlines $40M fund for facebook ipo glitches. `http://abcnews.go.com/blogs/business/2012/06/nasdaq-outlines-40m-fund-for-facebook-ipo-glitches/`.

[2] Nasdaq's facebook glitch came from race conditions. `http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[3] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA*, pages 141–152, July 2009.

[4] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *ICSE*, pages 1084–1093, June 2012.

[5] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 338–349, June 2003.

[6] P. Godefroid and N. Nagappan. Concurrency at Microsoft: An exploratory survey. In *Wkshop. on (EC)2*, July 2008.

[7] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, pages 231–240, May 2008.

[8] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *ASE*, pages 439–442, Sept. 2008.

[9] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, July 2011.

[10] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, June 2011.

[11] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, pages 241–255, Oct. 2010.

[12] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, pages 221–236, Oct. 2012.

[13] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA*, pages 16–26, July 2007.

[14] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, pages 299–309, June 2012.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, Mar. 2008.

[16] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI*, pages 378–388, June 2011.

[17] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. of Symp. on Principles of Programming Languages*, pages 327–338, January 2007.

[18] S. Park, R. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE*, pages 245–254, May 2010.

[19] S. Park, R. Vuduc, and M. J. Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *ICST*, pages 51–60, Apr. 2012.

[20] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, July 2011.

[21] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, pages 465–475, May 2003.

[22] 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*, pages 160–174, Oct. 2010.

[23] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. In *Workshop on Text Mining*, page 525, Aug. 2000.

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

[25] L. Wang and S. Stoller. Runtime analysis of atomicity for multithreaded programs. *Software Engineering, IEEE Transactions on*, 32(2):93–110, Feb. 2006.

[26] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? — a comprehensive characteristic study on incorrect fixes in commercial and open source operating systems. In *ESEC/FSE*, pages 26–36, Sept. 2011.

[27] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, Mar. 2011.

[28] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *ICML*, pages 1105–1112, June 2006.