Toward a Methodology to Expose Partially Fixed Concurrency Bugs in Modified Multithreaded Programs†

To Tsui‡
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
torrey.tsui@my.cityu.edu.hk

Shangru Wu
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
shangru.wu@my.cityu.edu.hk

W.K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

ABSTRACT
Many multithreaded programs incur concurrency bugs. A modified version of such a program, which the exposed concurrency bug is deemed fixed, should be subject to further testing to validate whether the concurrency bug may only be partially fixed. In this paper, we present a similarity-based regression testing methodology to address this problem. It is based on the notions of similar execution contexts of events and bug signatures. To the best of our knowledge, it also presents the first regression testing technique that manipulates thread schedules using a similarity-based active testing strategy.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Reliability, Verification

Keywords
Regression testing, debugging, concurrency bugs, execution contexts, active testing, similarity measure, modified versions

1. INTRODUCTION
Program debugging is time-consuming, tedious, and important in developing software applications. After a concurrency bug (which is a kind of spurious program state) has been exposed in a program, a corresponding program repair to fix the concurrency bug should be carried out, which modifies the former program version, say P, into the latter modified version, say P'.

Based on the modification, a typical regression testing research study focuses on ensuring whether P' has not been adversely affected by the modification. However, an important task for (regression) testing is to validate whether the bug is indeed completely fixed in P'.

For sequential deterministic programs, such a program repair may be validated by a traditional testing methodology, which is outlined as follows: Suppose that P is a sequential program and executing P over a test case t has exposed a fault in it. To ease our reference, we call t as a bug-triggering test case. In the simplest form of such a methodology, a developer may execute P' over the same bug-triggering test case t, and checks whether the previous failures in the program output and/or previous errors in program states associated with the bug have been removed. To follow this methodology, the developer should make the test case t executable by P' and be able to both observe and assess whether the failures and/or the error states no longer exist.

Owing to non-determinism, executing a multithreaded program over the same test case multiple times may generate different execution traces. Such a program may incur concurrency bugs such as deadlocks and data races. Hence, to apply the above methodology, developers should consider the added dimensions on thread schedules and concurrency bugs. The scenario is further complicated by the change in code from P to P', which even for the same test case t, a thread schedule s applicable to P may become infeasible for P'. In some cases, there may be other existing or new thread schedules in P' that can trigger a concurrency bug by executing P over t. In other words, developers may not be able to effectively re-trigger the same concurrency bug by following the previously observed thread schedule s even the concurrency bug in P' is still not fixed.

This paper presents a new regression testing methodology to address the above problem. We firstly recall that each event e (e.g., a lock acquisition) in an execution trace can be indexed with an *execution context*, such as the instruction (denoted by *inst*) that generates the event together with the call stack (denoted by *ctx*) at the time *inst* is executed [2]. Like existing active testing techniques, when executing P over t, it extracts the sequence of execution contexts of events that are related to the concurrency bug exposed in that execution trace as the bug signature.

However, unlike existing work, when executing P' over the same bug-triggering test case t, our methodology guides developers to actively match the execution contexts of interesting events (e.g., memory accesses and locking operations) along the execution trace. Specifically, it aims to identify a sequence of execution contexts that is *similar* to the bug signature extracted above through similarity matching. After exposing a concurrency bug that is deemed similar to the bug signature, the methodology requests testers to inspect the reported concurrency bug to verify whether it refers to the same concurrency bug in P that developers originally intended to fix.

† This work is supported in part by Early Career Scheme and General Research Fund of Research Grants Council of Hong Kong (project numbers 125113 and 123512). W.K. Chan is the correspondence author. All correspondences should be sent to W.K. Chan at wkchan@cityu.edu.hk .

‡ Mr. To Tsui was an undergraduate student that the time to produce this piece of research.

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.

InnoSWDev'14, November 16-22 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3226-2/14/11...$15.00
http://dx.doi.org/10.1145/2666581.2666592

$P \in Programs$	$t \in Test\ cases$
$op \in Operations$	$Operations = \{read, wirte, lock, unlock\}$
$\tau \in Threads$	$obj \in Shared\ objects \quad ctx \in Call\ Stacks$
$e \in Events$	$= \langle \tau, inst, op, obj, ctx \rangle$
$s \in Schedules$	$= P(t) = \langle e_1, \dots, e_n \rangle$

Figure 1. Notations used in our model

The main contribution of this paper is twofold. First, to the best of our knowledge, this paper proposes the first similarity-based regression testing methodology that checks whether a concurrency bug has been fixed in part. Second, we present a supporting tool of the methodology to show the feasibility of the methodology.

The rest of the paper is organized as follows. Section 2 reviews the preliminaries of our work followed by the methodology to be presented in Section 3. Section 4 describes the experiment. Section 5 reviews the closely related work, followed by the conclusion of the paper in Section 6.

2. PRELIMINARIES

2.1 Events and Thread Schedules

In this section, we review the notations to be used in this paper, which are also summarized in Figure 1.

An event e is a quintuple $\langle \tau, inst, op, obj, ctx \rangle$, means that the event e is generated by a thread τ executing an instruction *inst* under a call stack *ctx*, which performs an operation *op* on a shared object *obj*. An operation *op* is a memory access operation such as reading from a memory location or writing to it, or is a lock acquisition or release operation. For ease of our reference, we denote the four attributes *inst*, *op*, *obj*, and *ctx* of an event *e* by *e.inst*, *e.op*, *e.obj*, and *e.ctx*, respectively. We further refer to *offset*(*inst*) as the offset of the instruction *inst* in the (image) object file. We assume that an offset is an integer value.

Each execution of the program P generates a sequence of events. A thread schedule $s = P(t) = \langle e_1, \dots, e_n \rangle$ is a global sequence of events induced by executing the program P over the test case t. We denote an execution of P over the test case t by $P(t)$.

Executing a multithreaded program P over the same test case t multiple times may generate different thread schedules. Thus, we denote a set of m thread schedules generated by $P(t)$ as $S = \{s_1, s_2, \dots, s_x, \dots, s_m\}$. Similarly, a set of n thread schedules generated by $P'(t)$, meaning to execute the test case t on the modified version P', is denoted as $S' = \{s'_1, s'_2, \dots, s'_n\}$.

In the set S, we assume that there is at least one ***buggy schedule*** s_x that exposes a concurrency bug. We assume that each concurrency bug is an instance of a ***bug pattern***, which is a specific sequence of events formed by a set of threads interleaving among them.

2.2 Bug Signature

Lu et al. [4] empirically found that many concurrency bugs in multithreaded programs can be each triggered by enforcing a short sequence of events, and the length of such a sequence is no more than four (4). We refer to such an event sequence in a buggy schedule s_x as an instance of ***bug signature*** in s_x. We denote a bug signature as $bSig(s_x) = \langle e_{x1}, e_{x2}, \dots, e_{xk} \rangle$, where $2 \le k \le 4$, and for each event $e_{xi} \in bSig(s_x)$, $e_{xi} \in s_x$, where $1 \le i \le k$. Note that a bug signature refers to a triggering sequence of events, and a bug pattern refers to a sequence of events that quantifies a kind of concurrency bug; their corresponding sequences of events may or may not be overlapping.

Many concurrency bug detectors (e.g., RaceFuzzer [9], CTrigger [8], Maple [11]) firstly predict a set of thread interleaving from a thread schedule, and then attempt to manipulate a follow-up execution to produce another thread schedule that exhibits *the same* predicted thread interleaving(s) and further check whether a real concurrency bug associated with one of these thread interleaving is triggered. For ease of the distinction from the kind of thread interleaving in our methodology, we refer to the above kind thread interleaving generated from the former thread schedule as an ***exact buggy interleaving***.

2.3 Partial Bug Fixing

Fixing a concurrency bug is a software development activity. It may however only correct such a bug in part, which means to change the original code to incur a new concurrency bug or only change the set of triggering conditions of the same bug to another set. (We note that in this paper, we do not consider a modification resulted in a semantic bug.) We refer to such a bug as a ***partially-fixed concurrency bug***. We refer to the code patch to P with the aim of removing the exposed concurrency bug as a ***bug fixing patch*** (or a ***patch*** for short).

3. METHODOLOGY

3.1 Overview

In this paper, we consider partially-fixed concurrency bugs that can be triggered by the same test case. Suppose that a concurrency bug in P is not completely fixed in P'. Then, there should be at least one buggy schedule $s'_x \in S'$ that can trigger this partially-fixed concurrency bug. Our goal is to develop a methodology to find such a buggy schedule s'_x.

Note that in the worst case, testers may require to enumerate all possible thread schedules associated with $P'(t)$ in order to find s'_x. It is not what software testing aims for. We suppose that testers are given a test budget of n runs to check whether a concurrency bug is only partially-fixed (where n is much smaller than the number of thread schedules available). If all the test budgets have been exhausted, the checking procedure fails.

We observe that fixing a concurrency bug in a real-world program seldom drastically changes the behavior of the same program. We thus conjecture that a bug signature in s_x is likely to be *similar* to a bug signature in some but unknown test schedules s'_x of $P'(t)$ (i.e., $bSig(s'_x) \approx bSig(s_x)$ for some s'_x). We propose to use this observation to expose partially-fixed concurrency bugs in a modified version of a program.

Our **methodology** is iterative in nature. Given a program P and a test case t, an existing dynamic concurrency bug detector D firstly reports a real concurrency bug in $P(t)$. Along with this concurrency bug, there is a set of *exact buggy interleavings*. The bug signature of this concurrency bug is extracted from these buggy interleavings based on bug patterns. A developer then repairs the program P with the attempt to fix this particular concurrency bug, which results in a modified version of the program, denoted as P'. Then, the methodology goes into an iterative process until all the given test budgets on checking this

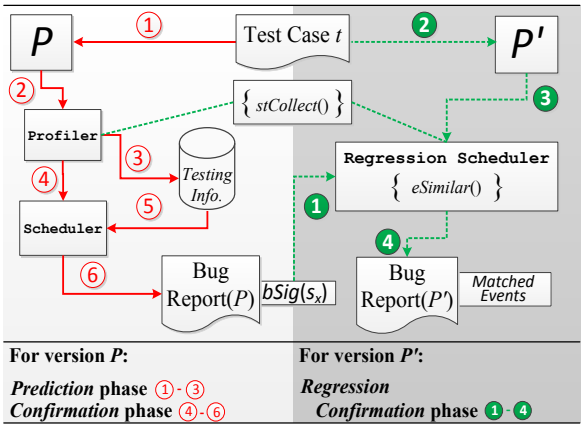

Figure 2. Major components of our methodology

```
1   Input: bSig(s_x) = ⟨e_x1, e_x2, ..., e_xk⟩ – bug signature
2   Input: P' – modified program version of P
3   Input: w – offset range
4   Input: d – maximum edit-distance
5   Input: s' = P'(t) = ⟨e'_1, e'_2, ...⟩ – an execution of P'(t)

6   potential interleaving ρ := ∅
7   i := 1
8   for each e' ∈ s' do
9   | if eSimilar(e', e_xi) then // e_xi ∈ bSig(s_x)
10  | | if i = 1 then
11  | | | obj := e'.obj
12  | | | ρ := ρ ∪ {e'}
13  | | | i := i +1
14  | | else if i > 1 && e'.obj = obj then
15  | | | ρ := ρ ∪ {e'}
16  | | | i := i +1
17  | | | explore(ρ)  //active testing
18  | | | if i > k then // k is the number of events in bSig(s_x)
19  | | | | break
20  | | | end if
21  | | end if
22  | end if
23  end for

24  Function eSimilar(e, e')
25  | if |offset(e.inst) − offset(e'.inst)| ≤ w &&
26  |      e.op = e'.op &&
27  |      ED(e.ctx, e'.ctx) ≤ d then
28  | | return true
29  | else
30  | | return false
31  | end if
32  end function
```

Algorithm 1. Regression Scheduler

patch has been exhausted. In each such iteration, it requires testers to provide parameters on a maximal extent of dissimilarity between $bSig(s'_x) \approx bSig(s_x)$ for some unknown s'_x. It then runs P' over t, and matches the events along the execution against the events in $bSig(s_x)$ in sequence based on the above inputted maximal extent of dissimilarity. We refer such a successful match as a ***s-match***. If the current execution trace induced by the execution does not expose a concurrency bug, it reports the matched events in the trace (which are similar to those in $bSig(s_x)$) for testers to manually adjust the parameters needed in the subsequent iteration(s). On the other hand, if a concurrency bug $bSig(s'_x)$, which is similar to $bSig(s_x)$, is exposed, the methodology requests testers to inspect the code to confirm whether they refer to the same concurrency bug.

3.2 Tool Support

Our methodology is currently supported by a system of four components: *profiler*, *scheduler*, *stCollect*, and *regression scheduler*. In Figure 2, the solid paths show the design of typical existing dynamic concurrency bug detection tools (e.g., Maple [11]), and the dotted paths show the new part of our methodology.

We have presented in the last sub-section that our methodology requires an existing detector D to expose a concurrency bug. This detector is implemented as the two components *profiler* and *scheduler*. To identify the execution contexts, we collect the call stack information of each event at runtime (denoted by the function *stCollect*()). This function would be invoked by both the components *profiler* and *regression scheduler* in order to match the execution contexts of events among different execution traces.

To verify whether the exposed concurrency bug is only partially fixed, we use *regression scheduler*. The regression scheduler (shown as green filled step 1 to step 4 in Figure 2) actively manipulates the execution with the aim of generating an event sequence similar to the sequence represented by $bSig(s_x)$ (See Section 3.4).

We refer to our tool as RegressionMaple because it is developed on the Maple framework [11]. We further note that if the ≈ operator is strengthened into an identity operator =, then RegressionMaple is degenerated into Maple.

In the rest of this section, we are going to present how we determine a pair of similar events (i.e., $e' \approx e$) and how we identify the thread schedules to be explored.

3.3 Similar Events

In our methodology, two events e' and e are similar if e' and e are only different up to a certain limit, which is controlled by two user-provided parameters w and d.

The parameter w specifies the offset range for instruction matching. The similarity between the two offsets in two given events e and e' is defined as the difference od between the two offsets: $od = |offset(e.inst) - offset(e'.inst)|$.

The parameter d indicates the maximum difference in the distance in call stack matching. The similarity between the two call stacks in two given events e and e' is denoted by $ed = ED(e.ctx, e'.ctx)$, where ED means an edit-distance. In our tool, we use the *Wagner-Fischer edit-distance* [7] to measure the distance between a pair of call stacks.

Thus, if we have $od \le w$ and $ed \le d$ and both events referring to the same kind of operation (i.e., $e.op = e'.op$), we say that the two events e and e' are similar, otherwise, not similar. This Boolean decision is denoted by function $eSimilar(e, e')$. The computation is shown in lines 24-32 in Algorithm 1.

Note that a modified version of a program may change thread usages including creation points, the number of threads, or their

(1) *Command*: **regression-maple active --target_iroot=146 --sw=10 --md=0 --target_image=pbzip2-part-fix --- ./pbzip2-part-fix -d -k -f -p2 zipped_file.tar.gz.bz2**

```
totsui2@prj1314c:~/FYP/1_benchmark/2_pbzip2/1_trigger$ regression-maple active -
-target_iroot=146 --sw=10 --md=0 --target_image=pbzip2-part-fix --- ./pbzip2-par
t-fix -d -k -f -p2 zipped_file.tar.gz.bz2
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

        # CPUs: 2
-------------------------------------------
        File #: 1 of 1
     Input Name: zipped_file.tar.gz.bz2
    Output Name: zipped_file.tar.gz

 BWT Block Size: 900k
     Input Size: 34823 bytes
Decompressing data...
    Output Size: 34334 bytes
-------------------------------------------

     Wall Clock: 0.380765 seconds
[REGSSION-MAPLE] === active iteration 1 done === (0.804443) (/home/totsui2/FYP/1
_benchmark/2_pbzip2/1_trigger)
[REGSSION-MAPLE] active threshold reached
```

(2) *Command*: **regression-maple active --target_iroot=146 --sw=5 --md=0 --target_image=pbzip2-part-fix --- ./pbzip2-part-fix -d -k -f -p2 zipped_file.tar.gz.bz2**

```
totsui2@prj1314c:~/FYP/1_benchmark/2_pbzip2/1_trigger$ regression-maple active -
-target_iroot=146 --sw=5 --md=0 --target_image=pbzip2-part-fix --- ./pbzip2-part
-fix -d -k -f -p2 zipped_file.tar.gz.bz2
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

        # CPUs: 2
-------------------------------------------
        File #: 1 of 1
     Input Name: zipped_file.tar.gz.bz2
    Output Name: zipped_file.tar.gz

 BWT Block Size: 900k
     Input Size: 34823 bytes
Decompressing data...
    Output Size: 34334 bytes
-------------------------------------------
A:Source/pin/vm_u/vm_signal_impl_unix.cpp:DeliverFatalSignal:DeliverFatalSignal:
4668: assertion failed: 0

################################################################################
## STACK TRACE
################################################################################
addr2line -C -f -e "/home/totsui2/FYP/pin_62732/intel64/bin/pinbin" 0x0305117e9
0x030512371 0x03072cb50 0x03072d910 0x03072e417 0x0307375e0 0x03073845c 0x03073c
410 0x0307d247a 0x0307feb2c 0x7f61b44cc864
[REGSSION-MAPLE] === active iteration 1 done === (0.806657) (/home/totsui2/FYP/1
_benchmark/2_pbzip2/1_trigger)
[REGSSION-MAPLE] active fatal error detected
```

Figure 3. Screenshot of RegressionMaple

appearance order in an execution. Therefore, in our matching of similar events, we do not consider their thread IDs.

3.4 Regression Scheduler

Algorithm 1 shows the design of the *Regression Scheduler*. The algorithm accepts a bug signature $bSig(s_x)$, a modified program P', an offset range w, and a maximum edit-distance d as inputs. For each profiled event e', if we have $eSimilar(e', e_{xi})$, where e_{xi} is an event in $bSig(s_x)$ (lines 8-9), the event is deemed to have a chance to form a potential but not exact buggy interleaving ρ.

Specifically, if e' is similar to e_{xi}, and e' is the first encountered event which is successfully matched with e_{xi}, then the event e' is added into ρ (lines 10-13), which is initially empty. If e' is a matched event but it is not the first encountered event ever successfully matched with e_{xi}, the event e' should access the same shared object that some other previous events currently kept in ρ has accessed (lines 14-16); and thus, it is one step closer to the bug similar to $bSig(s_x)$. Otherwise the event is ignored. For those events in ρ, the algorithm actively manipulates the thread schedules by suspending or postponing the event for a while with the aim of achieving the potential thread interleaving (as existing simple active testing strategy does) in line 17.

3.5 Limitations

In this section, we present the heuristics that we have summarized from our experimental trials.

To expose a partially fixed bug, we suggest to actively narrow down the search space for thread scheduling in the iterative process of our methodology. In our methodology, the parameters w and d are manually provided to select and filter relevant bug signatures in P'. We suggest the use of a small w initially, and either (1) increase its value if no matched potential interleaving candidate could be found or (2) decrease its value if the explored interleaving ρ cannot trigger the targeted bug. We also recommend to keep $d = 0$ initially.

However, we conjecture that the value of d should be adjusted in different executions to cater for structure-changed versions. To adjust d, for a fixed value of w, testers may either (1) increase d if no matched potential interleaving candidate could be found or (2) decrease its value if the explored interleaving cannot trigger any targeted bug. We have not observed a good strategy to modify d yet. We leave it as a future work.

Our methodology requires a manual code inspection to verify whether the concurrency bugs exposed in P and P' refer to the same concurrency bug. We leave the support of this activity as a future work. It is interesting to address this test oracle problem.

4. EXPERIMENT

4.1 Implementation and Experimental Setup

As presented in Section 3.2, we have implemented a tool on top of Maple [11] to support our testing methodology, and named it as RegressionMaple. The tool extended Maple to compute the call stack information of each event at runtime using a call stack unwinding algorithm [2] (after having ported it to a 64bits platform, and included the information of the first call stack frame). Up to 5 stack frames were encountered in the experiment (bug #2). Therefore, our tool did not limit the size of a call stack frame. It used Maple as its *profiler* and *scheduler* components.

The computer system configuration of the experiment was a virtual machine which is equipped with a dual-core CPU (Intel Xeon X5560 @ 2.8GHz), 2 GB RAM, and a 64-bits architecture environment.

Our subjects were three bugs adapted from Yu et al. [11] through modifying them into partially fixed bugs by applying mutation

Table 1. The experiment results of Regression Maple

#	Benchmark	LOC	Original Bug Type	Maple [11]			Regression Maple		
				Avg. Time (s)	Avg. Its.	S. D. Its.	Avg. Time (s)	Avg. Its.	S. D. Its.
1	Shared Counter	44	Synthetic bug	3.22	4	0	1.34	2.8	1.54
2	MySQL #169	128	Extracted real bug	10.75	10.2	7.19	1.64	2.4	0.80
3	Pbzip2	1954	Real bug	37.35	26.5	24.8	3.36	2.9	1.14

operators to the corresponding bug fixing patches. RegressionMaple was evaluated using these three bugs with two versions of them. One version is the version $P(t)$ which contains the original bug b obtained from Yu et al. [11]. Another version is the version $P'(t)$ which contains the partially fixed bug b' derived from b by applying one mutation operator on it.

In our experiment, we repeated RegressionMaple for 10 times for each partially fixed program version. In each attempt, we firstly applied Maple on $P(t)$ to obtain a bug triggering signature $bSig(s_x)$. Then, we used RegressionMaple to schedule $P'(t)$ with $bSig(s_x)$. In other word, each $bSig(s_x)$ used in different attempts may or may not be same. Please note that only the traces of $P'(t)$ were analyzed in the experiment. In all the 10 attempts, we only changed the value w and kept $d = 0$ according to our methodology. We leave the generalization of the experiment as a future work.

We also ran Maple on each of these 3 modified versions $P'(t)$ for 10 times. In each time, we ran Maple until the tool exposed the concurrency bug.

Figure 3 shows a screenshot of our tools that uses our methodology to trigger a partially fixed concurrency bug.

4.2 Experimental Results

4.2.1 Statistics

Table 1 compares Maple and RegressionMaple in terms of the performance for triggering partially fixed concurrency bugs. The two columns show the ID and the program used. The third column "LOC" reports the number of lines of code for each program subject. The fourth column shows the kind of bug obtained from Yu et al. [11]. The fifth and eighth columns labeled with "Avg. Time (s)" compare the average time required (in second) to trigger the partially fixed bug. The "Avg. Its." and "S.D. Its." columns report the mean number of iterations and the standard deviations over the 10 times of repeated experiments, respectively, that the corresponding tools have spent in order to expose the partially fixed concurrency bug.

For Maple, the performance is calculated as the time needed in both the profiling and testing iterations needed to trigger the same bug in each modified version. Maple needs profiling on each modified version as it is unaware of the presence of the original version while testing the modified version.

For RegressionMaple, without the need of profiling, the performance is calculated by the time needed for the testing iterations needed to trigger the bug. This tool does not need to include the profiling time on the modified version because the corresponding profiling has been done in the original version of each benchmark.

From the results presented in Table 1, we observed that RegressionMaple is more effective than Maple in revealing the partially fixed concurrency bugs.

Table 2 shows the sequence of w values used in the experiment on each modified version of each benchmark. In the table, the column "Run" shows the index of the test run, and the column "Iteration" shows the number of iterations performed in corresponding test run until exposing the partially fixed concurrency bug. Each cell shows the w value that is used for the corresponding run in the corresponding iteration. Moreover, the corresponding cell is highlighted when the run in that iteration successfully triggered the partially fixed concurrency bug. The w value is also underlined when it results in an *s-match*.

The table shows that the value for w can be set to either a small value or a large value, but not some value in between. Hence, it seems that one may start with a small value for w; and if some number of attempts fails, testers may switch to use a large value of w as a starting point. Nonetheless, this strategy may increase the cost (test runs or time) needed to expose the concurrency bug.

After an iteration is completed, a change in w based on the previous iteration data is required.

Table 2. The sequence of *w* values of RegressionMaple used in the experiment for the three benchmarks.

Benchmark #1 – Shared Counter

Iteration	Run									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
#1	10	50	0	100	10	5	1000	500	100	1000
#2		25	10	50		10	20	100	10	50
#3		15		13			10	5		5
#4								40		25
#5								20		12
#6								8		

Benchmark #2 – MySQL #169

Iteration	Run									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
#1	0	0	20	50	100	30	40	3	10	67
#2	50	10	60	60		60	80	30	70	
#3	100	100	100	80		90		80		

Benchmark #3 – Pbzip2

Iteration	Run									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
#1	0	10	20	0	10	7	50	14	21	0
#2		5	10	0	5	4	10	7	7	
#3		3	5	0	2	1	5	3	2	
#4			0				2			
#5							0			

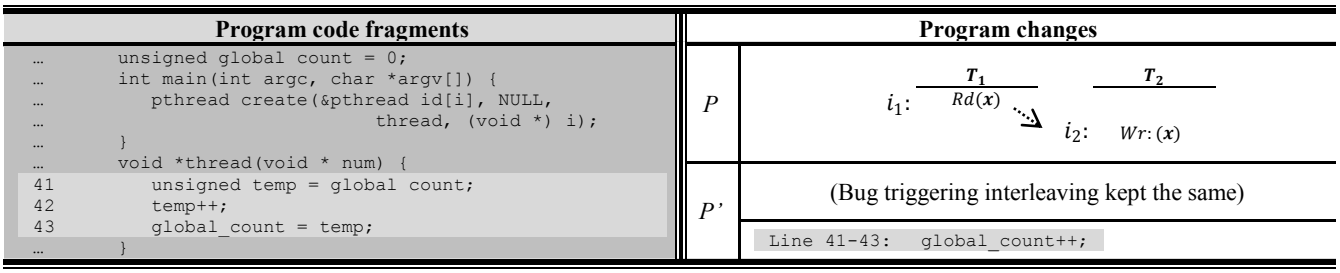

Figure 4. Concurrency bug in Shared Counter

In our experiment, we have utilized two kinds of operations, namely, ***jump*** and ***cut***. A *jump* substantially increases the *w* value to more than double or decreases the *w* value by more than half; and a *cut* moves *w* gradually. A *jump* provides a chance to drastically narrow down the *w* range for consideration. On the other hand, a *cut* can steadily narrow down the range.

A *boundary* is referred to the range of *w* tested so far. We noticed that there is still no exact criterion to determine whether a boundary is narrowed enough.

We use *jump* in a subsequent iteration if the boundary is large in the current iteration. If no s-match is detected (with *w=0*), *jump* increases *w* to, say, *w=100*. On the other hand, if an s-match is detected but the boundary is too large (e.g., $0 \le w \le 1000$), *jump* decreases *w* to, say, *w=20*.

If a boundary is narrowed enough, a *cut* should be applied: If an iteration reports an s-match with a narrow boundary, say $0 \le w \le 50$, we modify *w* by *cut*, say, to *w=25*. In our experiment, *cut* operation was heavily used.

The rest of this section describes our experiment in details.

4.2.2 Wrong Atomic Code Block – bug #1

Figure 4 shows the changes in the program code for the bug #1. The concurrency bugs in both versions (*P* and *P'*) of Shared Counter are the same. However a set of statements (i.e., line 41 to line 43) in bug #1 are combined as one statement in *P',* which demonstrated a wrong atomic statement assumption. We found that the bug can be triggered with a small w value ($7 \le w \le 15$).

In run #8, we used a combination of *jump* and *cut* operations (see Table 2). In the iteration #1, the initial *w* value was too large, i.e., *w=500*, even though there was an *s-match*. Therefore, a *jump* operation was selected, and we picked a much smaller value, i.e., *w=100* for the iteration #2. However, the iteration #2 still faced a similar situation, and thus, we performed a *jump* for the iteration #3 (*w=5*). We found that the result was different from previous iterations. Nonetheless, the iteration #3 reported a scheduling failure (i.e., thrashing). In this case, this *jump* operation was wasted and the boundary kept *5<w≤100*. As a result of the previous failed *jump*, the iteration #4 replaced this *jump* operation by a *cut* operation and found an s-match. Specifically, the iteration #4 *cut* the boundary *5<w≤100* roughly by half by a *w~50* value and resulted in a smaller boundary *5<w≤40*. Then, the iteration #5 used a *cut* operation because the boundary was narrow enough. We applied *w=20*, found an s-match, and thus further narrowed down the boundary into *5<w≤20*. The iteration #6 performed a *cut* and triggered the concurrency bug. This run demonstrated a failed *jump* operation (shown in iteration #3) and the further action to replace the *jump* operation by a *cut* (shown in iteration #4). This run consumed six iterations as a whole (and there were other runs consumed fewer iterations as shown in Table 2). Our finding shows that a better *w* value selected by a *jump* operation can save plenty of time.

4.2.3 Bug Pattern Changed – bug #2

Figure 5 shows the changes in the program code for the bug #2. The bug #2 is a multi-variables bug in *P*, but is mutated to a single-variable bug in *P′*. In the experiment, we found that three (3) iterations were usually enough to trigger the bug in *P'*(*t*) by RegressionMaple. We further performed additional attempts to ease our data analysis. We observed that for those attempts having $w \ge 63$, RegressionMaple always selected the same sequence of interleaving events as the original ones (whose existed in *P*(*t*)) to execute, and therefore triggered the bug (easily).

For the bug #2, we found that a *jump* operation can always trigger the bug in our experiment, because a normal *jump* substantially modifies *w*, e.g., *w=100*, and therefore, the modified *w*, can readily meet the threshold ($w \ge 63$) which triggers the concurrency bug.

We noticed that the above situation may not reflect a real program having a large number of shared objects. It is because for such a program, a large *w* may easily yield an *s-match* with unrelated events. Therefore, to prevent a misleading *jump* to be applied, testers may consider selecting a small *jump* whenever possible.

In run #3, the iteration #1 selected a small *w=20* initially. Because of the failure of *s-match*, in the iteration #1, we might perform a *jump* with *w=100* for iteration #2. However, we experienced to use a small *jump*. Specifically, we arbitrarily selected a value smaller than 100 but greater than 20, i.e., *w=60*. Nonetheless, the iteration #2 still could not detect any s-match. So, in the iteration #3, we applied a small *jump* with *w=100*. The iteration #3 detected a s-match and at the same time triggered the concurrency bug.

We noticed that even with same *w* value, some iterations derived from different runs, may have inconsistent results in terms of *s-match*. For example, both iteration #2 of run #1 and iteration #1 of run #4 used *w=50* value, but the former incurred a *s-match* and the latter did not. The cause of this inconsistent is the $bSig(s_x)$. Specifically, our experiment, in each run, applied Maple to obtain a $bSig(s_x)$ from *P* and then scheduled *P′* multiple times (i.e. iterations) by RegressionMaple with that $bSig(s_x)$. This could result in different $bSig(s_x)$ to be applied across the runs ,but could provide relatively more accurate data to our experiment. This circumstance was reflected in our example. The two $bSig(s_x)$

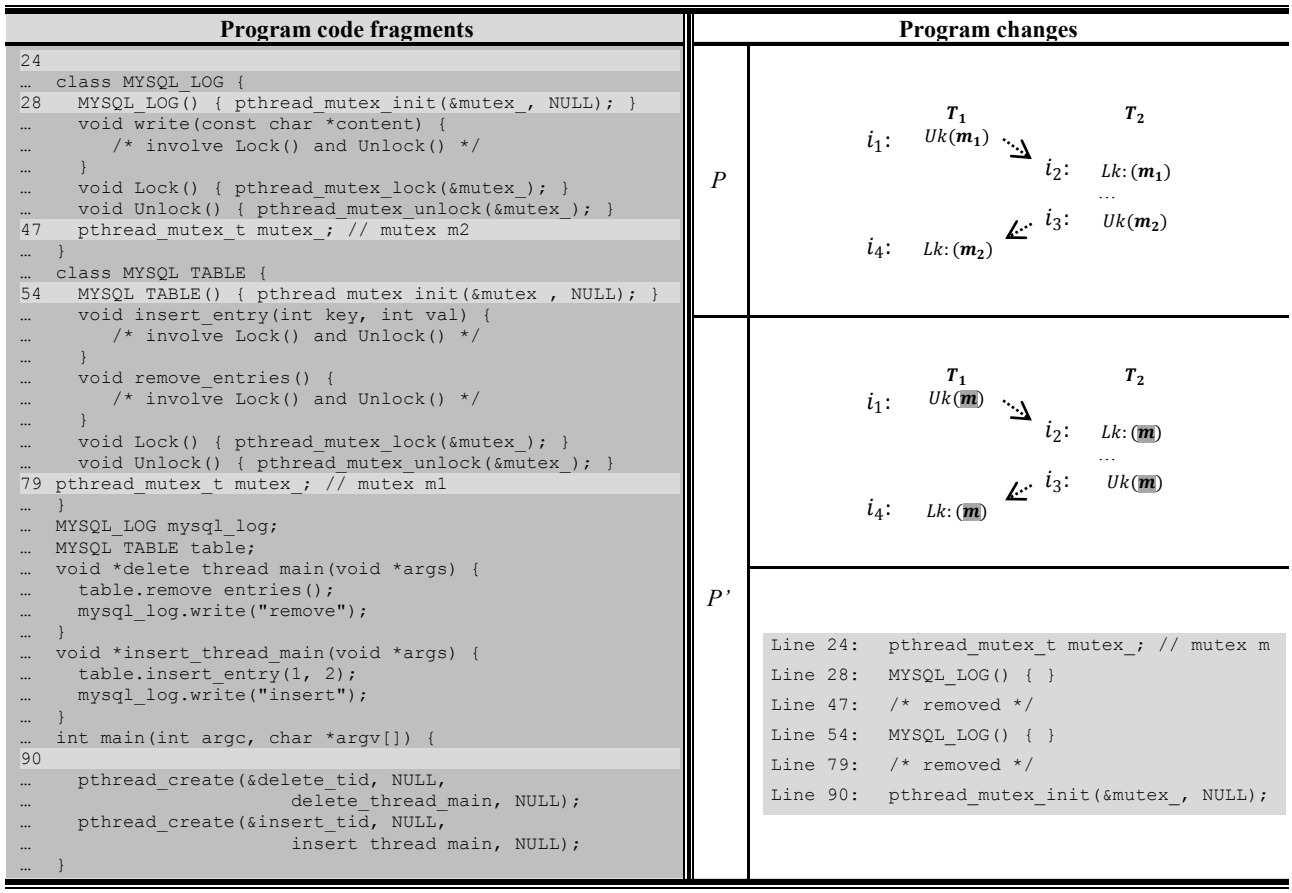

Figure 5. Concurrency bug in MySQL #169

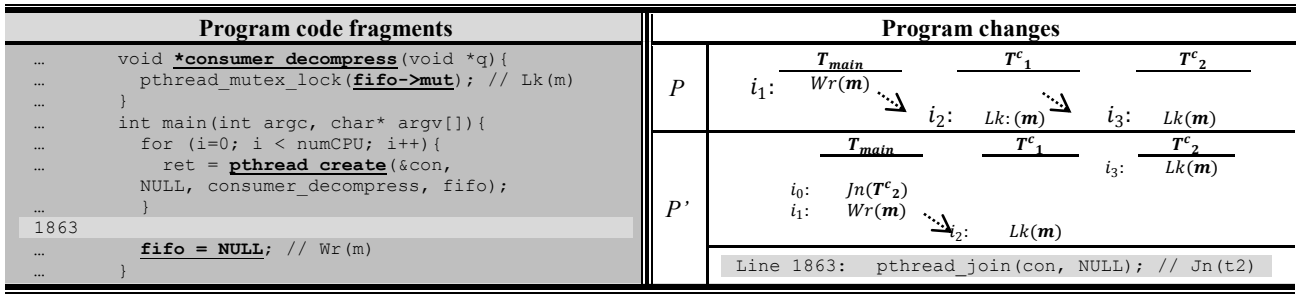

Figure 6. Concurrency Bug in Pbzip2

obtained from *P* in run #1 and run #4 were different, and therefore their *w* values cannot compare with each other.

4.2.4 Partial Resolution – bug #3

Figure 6 shows the changes in the program code for the bug #3. T_{main} is the main thread, and $T^c{}_1$ and $T^c{}_2$ are two consumer threads. In *P'*, the line 1863 is added as a barrier of T_{main}. The interleaving for $T_{main} \rightarrow T^c{}_2$ (i.e., $i_1 \rightarrow i_3$) is blocked in *P'*, but the interleaving $T_{main} \rightarrow T^c{}_1$ (i.e., $i_1 \rightarrow i_2$) is not blocked.

We found that the bug triggering schedules in *P* located by Maple can occasionally trigger the bug. That is, executing a bug triggering schedule two times had roughly one time that successfully triggers the bug.

In the experiment, we also found that not all bug triggering schedules, after ported to *P'*, can still trigger the bug. Moreover, all the bug triggering schedules located by Maple had roughly 50% in our data to expose the bug in *P'*. We believe that this was due to the priority assigned to the two concerned threads in the program. In this case, manual inspection on the code to select a bug signature which can directly trigger the bug in *P* is required.

In the run #4 (see Table 2), a bug triggering schedule after ported to *P'* could occasionally trigger the concurrency bug. Initially,

the iteration #1 selected a small value *w=0* and detected a s-match. Because *w* was already 0, which could not be further narrowed, and there still was a test budget, the iteration #2 proceeded with *w=0* again. The situation continued for subsequent iterations. Finally, the concurrency bug was triggered in the iteration #3.

4.3 Threats to Validity

The main threat of this experiment is the benchmarks we used. We conducted the study on three programs. The sizes of these programs were small. They were not necessarily representative of other programs of larger scales.

Another threat to validity is the patches for these programs. Because both the bug #1 and bug #2 were not real bugs, the corresponding bug fixing patches for these two programs were produced and applied by us. Other developers may repair these two bugs with different patches. Although the patch of the bug #3 followed a corresponding bug report, complex industrial programs with different characteristics may change the code structure in the course of fixing.

The test cases of these benchmarks were also a threat to validity. We used the test cases obtained from Yu et al. [11]. Other test cases may expose the same bugs with different execution traces in these programs.

We manually selected the values for *w*, and gradually increased or decreased the values based on the observed experimental data (e.g., whether the boundary was narrowed enough and whether there was any s-match). Other experiments may use other values. To improve the replicability of our experiment, we reported in Table 2 the sequence of *w* values used in each run in this paper.

5. RELATED WORK

In concurrent testing, there have been many work proposed for concurrency bug detection (e.g., [1][9][10][13]). Yu et al. [11] used memorization to increase interleaving coverage. Park et al. [8] exploited the observation from stress testing to prioritize suspicious schedules by their occurrence probabilities. Musuvathi et al. [5][6] attempted to address the scalability problem of large concurrent programs in systematic testing. In our methodology, we explore the notion of similar interleaving adaption in changed programs.

In the area of regression testing, Yu et al. [12] extended the technique of Deng et al. [3] and proposed SimRT that selected and prioritized potential bug-triggering test cases by coordinating program changes with respect to concurrency bugs. Their technique can greatly reduce the time cost in testing changed programs. Our work studies similar interleaving as a way to map execution points (if existed) across executions from different versions of the same program.

6. CONCLUSION

This paper introduced a similarity-based regression testing methodology to coordinate suspicious threads interleavings between two program versions. We have introduced a tool, RegressionMaple, to support the methodology. We have also illustrated the potential of bug interleaving adaption across versions in software evolution. Our experiment is limited and has not considered the possible impact on the use of the call stack similarity. Further generalizations of the methodology and the experiment should be performed so that the methodology may be applied to solve real-world regression testing problems.

REFERENCES

[1] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. ASPLOS'10*, pp. 167-178, 2010.

[2] Cai, Y., Chan, W. K., and Yu, Y. T. Taming deadlocks in multithreaded. In *Proc. QSIC'13*, pp. 276-279, 2013.

[3] Deng, D., Zhang, W., and Lu, S. Efficient concurrency-bug detection across inputs. In *Proc. OOPSLA '13*, pp. 785-802, 2013.

[4] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS'08*, pp. 329-339, 2008.

[5] Musuvathi, M. and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. PLDI '07*, pp. 446-455, 2007.

[6] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., and Neamtiu, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. OSDI '08*, pp. 267-280, 2008.

[7] Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv.* 33(1), pp. 31-88, 2001.

[8] Park, S., Lu, S., and Zhou, Y. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS'09*, pp. 25-36, 2009.

[9] Sen, K. Race directed random testing of concurrent programs. In *Proc. PLDI '08*, pp. 11-21, 2008.

[10] Sorrentino, F., Farzan, A., and Madhusudan, P. PENELOPE: weaving threads to expose atomicity violations. In *Proc. FSE '10*, pp. 37-46, 2010.

[11] Yu, J., Narayanasamy, S., Pereira, C., and Pokam, G. Maple: a coverage-driven testing tool for multithreaded programs. In *Proc. OOPSLA '12*, pp. 485-502, 2012.

[12] Yu, T., Srisa-an, W., and Rothermel, G. SimRT: an automated framework to support regression testing for data races. In *Proc. ICSE'14*, pp. 48-59, 2014.

[13] Zhang, W., Sun, C., and Lu, S. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proc. ASPLOS'10*, pp. 179-192, 2010.