SimRT: An Automated Framework to Support Regression Testing for Data Races

Tingting Yu, Witawas Srisa-an, and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska-Lincoln, USA
{tyu,witty,grother}@cse.unl.edu

ABSTRACT

Concurrent programs are prone to various classes of difficult-to-detect faults, of which data races are particularly prevalent. Prior work has attempted to increase the cost-effectiveness of approaches for testing for data races by employing race detection techniques, but to date, no work has considered cost-effective approaches for re-testing for races as programs evolve. In this paper we present SIMRT, an automated regression testing framework for use in detecting races introduced by code modifications. SIMRT employs a regression test selection technique, focused on sets of program elements related to race detection, to reduce the number of test cases that must be run on a changed program to detect races that occur due to code modifications, and it employs a test case prioritization technique to improve the rate at which such races are detected. Our empirical study of SIMRT reveals that it is more efficient and effective for revealing races than other approaches, and that its constituent test selection and prioritization components each contribute to its performance.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Tracing*; D.4.1 [Operating Systems]: Process Management

General Terms

Reliability, Experimentation

Keywords

Testing, Concurrency, Processes, Data Races, Kernels

1. INTRODUCTION

The advent of multicore processors has greatly increased the prevalence of concurrent software. However, failures due to concurrency faults still occur prolifically in deployed concurrent systems [56, 58, 59]. There are several classes of concurrency faults, including data races, atomicity violations, and deadlock. Among these, data races occur particularly frequently. Research has shown, for example, that data races remain a primary cause of concurrency faults in the Windows kernel [10]. Searches of bug repositories for widely used software systems such as the Linux OS and Apache Tomcat also reveal frequent occurrences of data races [22, 41].

In practice, software testing is the primary approach used by engineers to detect concurrency faults [27]. In software testing, the most typical approach for verifying test results involves checking system outputs following test execution. However, data races do not always produce failures that are visible in program outputs; therefore, techniques that inspect outputs to detect failures can be ineffective at detecting data races.

To address this problem, two general approaches have been used. First, developers can apply dynamic analysis techniques to detect potential races instead of waiting for the effect of races to propagate to output [2, 40]. Unfortunately, many reported potential races are false positives that cannot result in actual races. Furthermore, these techniques monitor every shared memory access and synchronization operation so they incur significant runtime overhead. A second group of techniques attempt to distinguish real races from potential races [41, 42]. These techniques explore multiple thread schedulings to determine whether a potential race can lead to a real race. To do this, the techniques may require a program under test to be executed several times under each test input in order to explore different thread interleavings.

When used with large test suites, the foregoing approaches can render the testing process unduly expensive, due to the high overhead of potential race detection and the costs of multiple program executions in race verification. Recent work [4] reports that each of the two approaches can introduce a 10x-100x slowdown for each test run. Such overhead increases as test suite size increases.

To address this problem, Yu et al. [54] propose MAPLE, which reduces the number of potential races that must be verified. Instead of verifying every detected potential race with a test input, MAPLE avoids re-verifying potential races that have previously been verified. This technique, however, still incurs a 10x-100x slowdown because it does not reduce the cost of detecting potential races.

While the challenges for testing concurrent programs for data races are extensive, an additional challenge arises as these programs evolve. As programs evolve they must be regression tested, to assess whether changes have adversely affected system behavior, and whether new code behaves as intended. Regression testing can be expensive, and to render it more cost-effective, various approaches have been suggested, including regression test selection and test case prioritization. However, most research on regression testing to date has focused on sequential software, and to our knowledge, none has considered the issues involved in regression testing for concurrency faults such as data races.

One approach for re-validating concurrent systems would be to

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.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568294

apply dynamic race detection techniques to new versions of systems following the application of traditional regression testing techniques [8, 36]. However, traditional regression testing techniques may not be cost-effective in this context. For example, empirical studies have shown that in certain cases, traditional regression test selection techniques can select inordinately large numbers of test cases, and yield no benefit in reducing the overhead associated with race detection [55].

To address this problem, in this work we propose SIMRT, an automated regression testing framework for use in detecting races that are induced in concurrent programs by code modifications. SIMRT identifies variables that can be accessed by multiple threads in a modified program, and that are impacted by modifications. SIMRT then employs a regression test selection technique to select the test cases from the program's regression test suite that exercise these shared variables in a manner that involves more than one thread. It is these test cases that are specifically relevant to detecting races.

While regression test selection targeting shared variables helps SIMRT reduce the cost of regression testing, the testing process is still unnecessarily burdened by the cost of dynamic race detection approaches, because different inputs tend to repeatedly execute the same memory locations and interleavings. Deng et al. [4] observe that up to 88% of test inputs always execute the same shared memory locations and interleavings and thus do not increase race detection rate as test cases execute. Thus, SIMRT next applies a greedy test case prioritization algorithm to schedule the test cases selected in its prior phase in an order that detects races faster.

To assess SIMRT, we conducted an empirical study in which we applied the approach to modified versions of nine concurrent source code objects, all of which contain data races that are the result of code modifications. We assessed both the regression test selection and test case prioritization components of our approach by comparing SIMRT to traditional and baseline regression test selection and test case prioritization techniques. Our results show that SIMRT is more efficient than a common baseline regression test selection technique, and substantially more efficient than the default technique of executing all test cases, while retaining the effectiveness of those techniques. Our results also show that the prioritization of test cases by SIMRT substantially increases the rate at which races are detected with respect to a common baseline prioritization technique, and a random ordering of test cases.

2. BACKGROUND

2.1 Regression Testing

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, but reusing all of T (the *retest-all approach*) can be inordinately expensive. Thus, a wide variety of approaches have been developed for rendering reuse more cost-effective via regression test selection and test case prioritization ([52] provides a recent survey).

Regression test selection (RTS) techniques select, from test suite T, a subset T' that contains test cases that are important to rerun. In our own prior work [36], we presented the RTS technique DEJAVU. DEJAVU performs simultaneous depth-first traversals on control flow graphs (CFGs) for procedures in P and P' to find *dangerous edges* that lead to code that has changed. Execution traces of test cases (bit vectors indicating whether basic blocks were covered) on P are then used to select test cases that traversed dangerous edges in P. We have shown [35] that when certain conditions are met, DEJAVU is *safe*; i.e., it cannot omit test cases which, if executed on P', would reveal faults in P' due to code modifications.

Test case prioritization (TCP) techniques reorder the test cases in T such that testing objectives can be met more quickly, and one potential objective involves revealing faults. A wide range of TCP techniques have been proposed and studied, and one technique that has proven successful is the "additional-block-coverage" technique [37]. Given the results of executing tests and gathering trace information, this technique prioritizes test cases in terms of the numbers of new (not-yet-covered) basic blocks they cover, by iteratively selecting the test case that covers the most not-yet-covered blocks until all blocks are covered, then repeating this process until all blocks have been covered.

Because TCP techniques do not themselves discard test cases, they can avoid the drawbacks that can occur when regression test selection cannot achieve safety. Alternatively, in cases where discarding test cases is acceptable, test case prioritization can be used in conjunction with regression test selection to prioritize the test cases in the selected test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficially than if test cases were not prioritized.

A key insight behind the techniques just described is that certain testing-related tasks (such as gathering code coverage data) can be performed in the "preliminary period" of testing, before changes to a new version are complete. The information derived from these tasks can then be used during the "critical period" of testing after changes are complete and when time is more limited.

2.2 Race Detection and Verification

A data race occurs when two threads can simultaneously access a shared variable, with at least one access being a write [40]. Many static [9, 48] and dynamic [11, 29] analysis techniques have been developed to detect data races. In this work, we consider dynamic race detection techniques.

Algorithms to dynamically detect data races can be classified as lock-set [40, 60], vector-clock [2, 11], and hybrid approaches [29, 42]. These algorithms can report false positives, because they cannot guarantee that a race can really occur under specific thread interleavings. We refer to the races reported by these algorithms as *potential races*. Techniques have also been proposed to determine whether potential races identified by a detector can actually occur. We refer to races that have been verified in this way as *real races*. However, even race verifiers cannot differentiate harmful races from benign races; they report any pair of unsynchronized accesses (at least one of which is a write) as a race.

RACEFUZZER [41] is a race detection tool that combines dynamic race detection and race verification. RACEFUZZER computes pairs of program instructions that could *potentially* race during concurrent execution. It then *randomly schedules* the program under test based on the computed instruction pairs to allow *real* racing events to be placed temporally next to each other.

Employing random schedules for race verification is precise andcost effective, but can be incomplete [41, 58]. Random scheduling does not guarantee that verification can distinguish all real races from potential races; rather, it guarantees that if it can cause a race to occur, this potential race will become a real race. On the other hand, complete replay techniques such as PENELOPE [45] may incur much higher overhead due to context switches.

2.3 Testing for Races

Software testing requires test inputs. Given a set of test inputs, the race detection process for a multi-threaded program involves two steps. First, for each test input, the program is executed by a dynamic race detection tool that identifies potential races (race

```
1  public boolean containsValue(Object value) {
2          Entry tab[] = table; //block ID: 1
3          if(value == null){
4             for (int i=tab.length; i--> 0;) //block ID: 2
5                                ...
6          }
7          else {
8                          ...
9                          return true; //block ID: 4
10         }
11         return false;
12 }
13
14
15
16
17
18
19
20
21 public void clear() {
22         ...
23         synchronized(this){
24         Entry tab[] = table; //block ID: 3
25         for (int i=0; i<tab.length; i++)
26            tab[i] = null; //block ID: 4
27         }
28         //block ID: 5
29         ...
30 }
```

```
1  public boolean containsValue(Object value) {
2          Entry tab[] = table;
3          if(value == null){
4             for (int i=0; i<tab.length; i++) //change
5                                ...
6          }
7          else {
8             ...
9             return containsNull(); //change
10         }
11         return false;
12 }
13
14 private boolean containsNull() {
15         Entry tab[] = table;
16         for (int i=0; i<tab.length ; i++)
17            for (Entry e=tab[i]; e!=null; e=e.next)
18               ...
19    }
20
21 public void clear() {
22         ...
23         synchronized(this){
24         Entry tab[] = table;
25         for (int i=0; i<tab.length; i++)
26                tab[i] = null;
27         }
28         modCount++; //change
29         ...
30 }
```

Figure 1: Original HM program P (left) and a modified version of the HM program P' (right)

detection). Second, for any input t that produces a set of potential races $PRaceSet$ in the first step, and for each $pr \in PRaceSet$, the program is executed n ($1 \leq n \leq N$) times to verify pr under t, where N is defined by users in terms of a testing budget (race verification). We define $N = 1$ for SIMRT to simulate a resource-constrained testing environment.

Our SIMRT approach uses RACEFUZZER to test for races by following the foregoing two steps. However, in the race verification phase, RACEFUZZER verifies every potential race reported in the race detection phase for the given input regardless of whether this race has already been verified by previous inputs. This can be expensive when given large test suites, particularly when each test case reports redundant potential races in the race detection phase. To alleviate this problem, we adjusted RACEFUZZER so that, once a potential racing pair is confirmed to be a real race, it is not verified again under another input. Specifically, for each test input, SIMRT first invokes RACEFUZZER and reports potential races. If there exist any potential races that have not been confirmed as real races, SIMRT invokes RACEFUZZER to verify each of them, under the same test input, before proceeding to the next test input. As such, the actual number of test runs for an entire test suite T is:

$$TR = |T| + \sum_{i=1}^{|PRaceSet|} NT_i$$

Here, $|T|$ is the total number of original tests, $|PRaceSet|$ is the number of potential races, and NT_i is the number of tests required to verify the i^{th} pair in $PRaceSet$.

3. APPROACH

Figure 1 contains an example that we use to illustrate our approach. The figure provides code snippets from two versions of the JDK's `HashMap` utility (slightly modified, and referred to here as `HM`). The variables `table`, `tab[i]`, `next` and `modCount` are identified as shared variables by thread escape analysis (see Section 3.2). A test driver for this code instantiates an `hm` object from class `HM`. The two parameters in the constructor of `HM` specify the initial capacity and load factor. Each of the test cases for the code involves two components, each executing in its own thread. We define four test case components:

case$_0$: hm.clear()

case$_1$: hm.containsValue(new HM(100, 10.0f));

case$_2$: hm.containsValue(new HM(0, 100.0f));

case$_3$: hm.containsValue(null)

A test case tc_i is denoted by (m, n) $(0 \leq m, n \leq 3)$, where m and n denote two of the foregoing cases (i.e., $case_m$ and $case_n$).

Suppose there are six test cases generated for version P of `HM`: $tc_1 = (0, 1)$, $tc_2 = (0, 2)$, $tc_3 = (0, 0)$, $tc_4 = (0, 3)$, $tc_5 = (1, 2)$, and $tc_6 = (3, 3)$. In version P' of `HM`, there are two changes that cause three races to occur. The first change involves addition of a new method `containsNull()` and a call to it from line 9. This change causes a read-write race involving `tab[i]` at line 17 and `tab[i]` at line 26 when executing tc_1 or tc_2 (i.e., concurrently executing `containsNull()` and `clear()`). The second change involves addition of a new line of code (line 28) that reads and writes a class variable `modCount`. Two data races occur in this case, involving `modCount` when tc_3 is executed; one is a write-write race, and the other is a read-write race.

3.1 Overview of SimRT

Figure 2 provides an overview of SIMRT. SIMRT contains five components: *SVLocator*, *ImpAnalyzer*, *Matcher*, *Selector*, *Ranker*. Let P be a program, let P' be a modified version of P, let C' be the set of changes made to P to produce P' (a set of program locations in P'), let B'_{SV} denote a block in a method in P' that contains a shared variable SV, and let B_{SV} be a block in a method in P that corresponds to B'_{SV}. SIMRT first computes a list of shared variables L'_{SV} in P' using *SVLocator*. Next, *ImpAnalyzer* updates L'_{SV} by mapping B'_{SV} to B_{SV}, and identifying the shared variables in L'_{SV} that are potentially impacted by one of the changes C' in P'. Next, *Matcher* iterates over each SV in L'_{SV}, and selects the SVs that can be matched into pairs. The output of *Matcher* is a list of impacted shared variable pairs (I'_{SVP}) that are coverage targets for RTS and TCP techniques. Next, *Selector* selects $T' \in T$ for use in regression testing P'. Finally, *Ranker* prioritizes

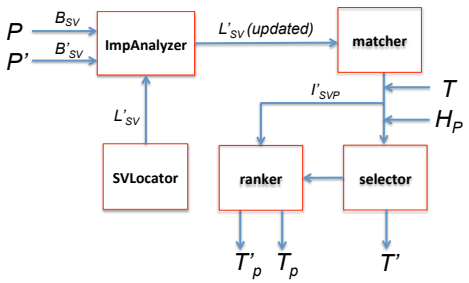


Figure 2: Overview of SimRT

the test cases in T' (or T), producing T'_P (or T_P). Both *Selector* and *Ranker* utilize coverage history information provided as H_P. We describe each of these components in the sections that follow.

3.2 Shared Variable Identification

SVLocator produces a list of variables L'_{SV} that can be accessed by multiple threads in P'. In Java, shared variables can be identified by using thread escape analysis [20, 38]; we adopted the conservative shared variable detection algorithm proposed in [16], that uses the `ThreadLocalObjectAnalysis` API [15] provided by SOOT [44] to compute variables that can potentially be read from and written to by multiple threads simultaneously. We also used the alias analysis [28] provided by SOOT to identify a set of escaping variables that can potentially access the same location.

Each shared variable (SV) is defined as a 7-tuple < $C.M$, N, D, L, A, B, I > where C is a class name, M is a method signature, N is the name of the SV, D is a memory access identifier (SVs that potentially access the same memory location share the same identifier), L is a line number at which the SV occurs, A denotes the access operation (read or write) performed on the SV, B is the block ID (a unique number in $C.M$) in P where SV is mapped to, and I is a boolean value indicating whether SV is impacted. Note that $C.M$ must exist in both P and P'. If a method M' in P' in which a SV occurs does not exist in P, we locate, in the call graphs for P', the method that most directly calls M' that also exists in P, and, if no such method is found, $C.M$ is defined as $\perp$.

In the `HM` program, shared variables in P' are displayed as the following initialized tuples:

```
SV1:<HM.containsValue(Object), table, 1, 2, read, ⊥, ⊥>
SV2:<HM.containsValue(Object), table, 1, 15, read, ⊥, ⊥>
SV3:<HM.containsValue(Object), tab[i], 2, 17, read, ⊥, ⊥>
SV4:<HM.containsValue(Object), next, 3, 17, read, ⊥, ⊥>
SV5:<HM.clear(), table, 1, 24, read, ⊥, ⊥>
SV6:<HM.clear(), tab[i], 2, 26, write, ⊥, ⊥>
SV7:<HM.clear(), modCount, 4, 28, read, ⊥, ⊥>
SV8:<HM.clear(), modCount, 4, 28, write, ⊥, ⊥>
```

Here, `<HM.containsValue(Object), table, 1, 15, read, ⊥, ⊥>` indicates that in line 15 in P', variable `table` is read, with access identifier 1. The method `HM.containsNull()` in which `table` occurs does not exist in P; thus, `HM.containsValue(Object)` is filled into $C.M$. Elements B and I are not specified until the comparison algorithm (Section 3.3) is invoked.

3.3 Shared Variable Impact Analysis

Figure 3 displays the algorithm used by IMPANALYZER. The algorithm takes program P, modified version P', and a list of shared variables L'_{SV} with initialized tuples as inputs, and returns the updated L'_{SV}. First, the algorithm constructs control flow graphs (CFGs) G and G' for all methods in P and P' (line 4). Each node in a CFG is represented using a unique block ID. Next, the algorithm compares each CFG G in P to the corresponding G' in P' by calling *Compare* (line 6). The comparison begins with entry nodes E and E'. Given two CFG nodes N and N', *Compare* determines whether N and N' have successors (S and S') whose code differs along pairs of identically labeled edges (lines 14-20, 23).

Up to this point, *ImpAnalyzer* behaves identically to the original DEJAVU algorithm [36]. However, *ImpAnalyzer* uses different mechanisms to handle node comparisons. *ImpAnalyzer* obtains a set of line numbers in block node S' (line 22). If the code associated with S and S' is the same (line 23), the algorithm iterates over L'_{SV}, and picks the SVs that are contained in S' (line 24-26). For each of these SVs, tuple element B is set to the block ID of S (line 27), and tuple element I is set to $false$ (line 28), indicating

```
procedure UpdateSVInfo
1: Inputs: P, P', L'_SV
2: Outputs: L'_SV for P' /*updated*/
3: begin
4: construct CFGs for P and P'
5: for each G ∈ CFG and G' ∈ CFG'
6:     Compare(E, E') /*start from entry nodes*/
7: endfor
8: return L'_SV
9: end

procedure Compare
10: Inputs: N, N' /*nodes in G and G'*/
11: Outputs: L'_SV for G' /*updated*/
12: begin
13: isVisted(N) = true /*whether N is visited*/
14: for each successor S of N ∈ G
15:     if (N, S) is labeled
16:         L = the label on edge (N, S)
17:     else
18:         L = ϵ
19:     endif
20:     S' = the node in G' such that (N', S') has label L
21:     if isVisited(S) is false
22:         LNS_S' = getBlockLineNumbers(S')
23:         if LEquivalent(S, S') /*if code is equivalent*/
24:             for each SV ∈ L'_SV
25:                 ln_SV = getSVLineNumber(SV)
26:                 if ln_SV ∈ LNS_S'
27:                     setOldBlockID(SV, S)
28:                     setIsImpacted(SV, false)
29:                     break
30:                 endif
31:             endfor
32:             Compare (S, S')
33:         else
34:             for each SV ∈ L'_SV
35:                 if isImpactedBy(SV, S')
36:                     setOldBlockID(SV, S)
37:                     setIsImpacted(SV, true)
38:                     break
39:                 endif
40:             endfor
41:         endif
42:     endif
43: endfor
44: end
```

Figure 3: ImpAnalyzer algorithm

that this shared variable is not impacted.

If the code associated with S and S' differs along some pair of identically labeled edges, *ImpAnalyzer* iterates over L'_{SV} and determines the impacted shared variables (line 34-35). If a shared variable SV is impacted by the change, tuple element B is set to old block ID S (line 36), and tuple element I is set to $true$ (line 37), indicating the existence of impact. A shared variable is considered impacted (line 35) in the following cases:

1. If a change to P' involves an SV in P' and this SV is either added to or modified in P', an SV is considered impacted.
2. If a change to P' involves adding or modifying a method, all SVs in this method are considered impacted.
3. If a change to P' involves a synchronization statement, all the SVs in the entire region of the synchronized block are considered impacted. For example, if line 23 in Program P is omitted, all shared variables inside the synchronization block (SV_5 and SV_6) are considered impacted. In addition to considering blocks using the synchronized keyword, we also consider blocks using explicit locks, including `lock()`, `unlock()`, and `trylock()`, and locks implementing a

ReadWriteLock interface (`readLock()`, `writeLock()`). We do not consider changes involving `wait()`, `notify()`, or `notifyAll()`; these must be called inside a synchronized block [14], and changing them does not affect SVs in synchronized blocks.

4. If a change to P' involves removing a volatile keyword in the declaration of an SV, all subsequent uses of this SV are considered impacted.
5. If a change to P' does not involve an SV but its full impact set in P' (obtained by traditional impact analysis such as static forward slicing) includes one or more SVs, these SVs are considered impacted.

In the example shown in Figure 1, IMPANALYZER compares the method `containsValue(Object)` in P and P'. The algorithm begins by visiting the node with block ID 1 in P (lines 2-3) and compares it to its corresponding node in P'. The two blocks are equal, so the algorithm updates tuple elements B and I for all SVs in lines 2-3 of P'. Thus, <HM.containsValue(Object), table, 1, 2, read, ⊥, ⊥> is set to <HM.containsValue(Object), table, 1, 2, read, 1, false>. Here, $B = 1$ is the block ID in P that `table` is mapped to, and $I = false$ indicates that `table` is not impacted.

Next, IMPANALYZER compares the nodes at line 4 in both P and P'. Although line 4 is changed in P', the change does not impact any shared variables. Thus, the comparison proceeds to the next node in P. When tIMPANALYZER visits the node corresponding to block 4 in `HM.containsValue(Object)` in P, it discovers that line 9 in P' is changed by addition of the call to method `containsNull()`. Thus, all SVs in this method are considered impacted. For example, <HM.containsValue(Object), next, 3, 17, read, ⊥, ⊥> is set to <HM.containsValue(Object), next, 3, 17, read, 4, true>.

After the algorithm returns, the two undefined elements in each SV are noted. In the example shown in Figure 1, the list of shared variables is updated as follows:

SV_1: <HM.containsValue(Object), table, 1, 2, read, 1, false>,
SV_2: <HM.containsValue(Object), table, 1, 15, read, 4, true>,
SV_3: <HM.containsValue(Object), tab[i], 2, 17, read, 4, true>,
SV_4: <HM.containsValue(Object), next, 3, 17, read, 4, true>,
SV_5: <HM.clear(), table, 1, 24, read, 3, false>,
SV_6: <HM.clear(), tab[i], 2, 26, write, 4, false>,
SV_7: <HM.clear(), modCount, 4, 28, read, 5, true>,
SV_8: <HM.clear(), modCount, 4, 28, write, 5, true>.

3.4 Matching Shared Variables

Because race detection involves pairs of shared variable accesses, the next step is to use *Matcher* to identify a set of impacted shared variable pairs I'_{SVP} serving as coverage targets for the *Selector* and *Ranker* modules. We define an impacted shared variable pair $ISVP$ as a pair of shared variables SV_i and SV_j, such that at least one of them is impacted, and at least one of them is a write access.

To illustrate, *Matcher* takes the updated shared variable list L'_{SV} computed in Figure 3 as input, and outputs the potential impacted shared variable pairs I'_{SVP}. For each shared variable SV_i in L'_{SV}, if SV_i is impacted (determined by querying element I in the SV_i tuple), the algorithm iterates over L'_{SV} to identify each SV_j in L'_{SV} that is shared with SV_i, regardless of whether SV_j is impacted. As a result, each qualified $ISVP$ = (SV_i, SV_j) is added to I'_{SVP}. In our example, five variables are impacted (SV_2, SV_3, SV_4, SV_7, and SV_8), but only SV_3, SV_7, and SV_8 qualify for pairing, so I'_{SVP} = <(SV_3, SV_6), (SV_7, SV_8), (SV_8, SV_8)>.

procedure *TestSelection*
45: **Inputs:** T, H_P, I'_{SVP}
46: **Outputs:** T'
47: **begin**
48: $T' = \phi$
49: **for** each $(SV_i, SV_j) \in I'_{SVP}$
50: $B1$ = `getOldBlockID`(SV_i)
51: $B2$ = `getOldBlockID`(SV_j)
52: **for** each $tc \in T$
53: **if** $H_P(tc, B1) = true$ and $H_P(tc, B2) = true$
—: and $H_P(tc, Tid_{B1}, Tid_{B2}) = true$
54: $T' = T' \cup tc$
55: **endif**
56: **endfor**
57: **endfor**
58: **end**

Figure 4: Regression test selection algorithm

3.5 Regression Test Selection

The next step is to use *Selector* to select test cases that are relevant to race detection. A naive approach is to select every test case that traverses any $(SV_i, SV_j) \in I'_{SVP}$. However, this may include test cases that execute shared variables but involve only one thread. Instead, SIMRT selects test cases by traversing the tuple elements B of SV_i and SV_j involved in different threads. To facilitate this, when collecting coverage information for each block in the original program P, which is done during the preliminary phase of testing, we also record the thread IDs that cover each block. As such, the constructed test history indicates (1) whether a block is covered, and (2) the IDs of the threads that cover this block.

Figure 4 shows the *Selector* algorithm. The algorithm takes three inputs: test suite T, the impacted shared variable pairs I'_{SVP}, and the test coverage history that indicates which test cases in T covered which statements in P with which thread. H_P is denoted by tc_i = <($C.M$, B, TID)>, where tc_i is the test case with number i, $C.M$ is the class name combined with the method signature, B is the block ID that tc_i covers, and TID is the ID of the thread that covers B. For each shared variable pair (SV_i, SV_j) in I'_{SVP}, the algorithm obtains the matching block IDs $B1$ and $B2$ for SV_i and SV_j (line 50-51) as coverage targets. Based on the coverage history H_P, the algorithm selects all test cases from T that traversed $B1$ and $B2$ (the first two conditions in line 53) with different threads (the third condition in line 53), and adds them to T' (line 54).

In our example, suppose the coverage history H_P holds the following coverage information for each of the six test cases:

tc_1 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1),
(HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,
tc_2 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1),
(HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,
tc_3 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1),
(HM.clear(), 3, 2), (HM.clear(), 4, 2), (HM.clear(), 5, 2)>,
tc_4 = <(HM.clear(), 3, 1), (HM.clear(), 5, 1),
(HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 2, 2)>,
tc_5 = <(HM.containsValue(Object), 1, 1), (HM.containsValue(Object), 4, 1),
(HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,
tc_6 = <(HM.containsValue(Object), 1, 1), (HM.containsValue(Object), 2, 1),
(HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 2, 2)>.

In this case, both tc_1 and tc_2 cover one target, (SV_3, SV_6), with different threads, and tc_3 covers two targets, (SV_7, SV_8) and (SV_8, SV_8), with different threads. Test cases tc_5, and tc_6 do not cover any targets. Test case tc_4 covers targets (SV_7, SV_8) and (SV_8, SV_8) but with only one thread. Therefore, T' = <tc_1, tc_2, tc_3>. In contrast, most traditional RTS techniques would select all six test cases for T' because they all cover changed blocks. By running

tests tc_1, tc_2 and tc_3 we can detect all three races, while tc_4, tc_5 and tc_6 do not help expose races.

Note that we select both tc_1 and tc_2 even though they cover the same target. The reason for this is because covering (executing) the two shared variables in the target pair under one test input is not necessarily sufficient to determine whether they access the same memory addresses [57]. Different inputs could cause different states to exist in the same code region, causing two instructions in the target pair to access different memory locations under one input and the same memory location under a different input.

3.6 Test Case Prioritization

Ranker prioritizes test cases, beginning with those identified by *Selector*. *Ranker* iterates over T', selects the test case tc_i that covers the most impacted shared variable pairs ($ISVP$s) in I'_{SVP}, and places tc_i in T'_P. Next, *Ranker* iteratively selects the test case tc_i that covers the most $ISVP$s and appends tc_i to T'_P. When multiple test cases cover the same number of $ISVP$s, *Ranker* chooses one randomly. If each $ISVP$ has been covered by at least one test case, and the remaining unprioritized test cases cannot add additional coverage, *Ranker* resets the coverage vectors for all unprioritized test cases to their initial values, and reapplies the prioritization approach, ignoring previously prioritized test cases.

In our example program, tc_1 and tc_2 cover (SV_3, SV_6) with different threads, and tc_3 covers (SV_7, SV_8) and (SV_8, SV_8) with different threads. *Ranker* first places tc_3 in T_P. Because tc_1 and tc_2 achieve the same additional coverage, the algorithm randomly selects one of these (say, tc_1) and appends it to T'_P. Next, the algorithm resets the coverage data. As such, tc_2 becomes the test case that covers the most $ISVP$s and is appended to T'_P. Therefore, the output of *Ranker* is T'_P = <tc_3, tc_1, tc_2>. When running T'_P, all three races can be exposed by the first two test cases.

As noted in Section 2, the safety of RTS techniques depends on certain conditions, and these include the condition that the program under test be executed deterministically. This condition cannot generally be met for the class of programs that we consider, and thus, SIMRT may omit test cases from T' that could reveal races in P'. In this context, it is important to note that even a retest-all approach can "omit" race-revealing test cases, because a given test case may or may not expose a fault in a given run when nondeterminism affects thread behavior. The motivation for selecting a subset of test cases, however, is to select test cases that are more likely to reveal races than others, allowing testers to focus on more worthwhile pursuits. Nevertheless, if test engineers wish, they can use *Ranker* to further prioritize the remaining test cases ($T - T'$), and execute those test cases as well. This process is performed using the approach just described, applied to $T - T'$, in two steps, where step 1 focuses on test cases that cover the most impacted shared variables, and step 2 focuses on test cases that cover only non-impacted shared variables.

In our example, when prioritizing the remaining test cases (having already set T_P to <tc_3, tc_1, tc_2>), *Ranker* notes that (step 1) tc_4 covers two impacted shared variables, SV_7 and SV_8, and tc_5 covers three impacted shared variables, SV_2, SV_3, and SV_4, and appends these to T_P, obtaining <tc_3, tc_1, tc_2, tc_5, tc_4>. In step 2, since tc_6 covers SV_1, a shared variable that is not impacted, *Ranker* appends that to T_P. Ultimately, T_P = <tc_3, tc_1, tc_2, tc_5, tc_4, tc_6>.

4. EMPIRICAL STUDY

We wish to determine whether SIMRT is cost-effective, and ideally such an assessment would involve comparisons with existing state-of-the-art approaches for detecting races in modified software. There are, however, no existing approaches that have this specific goal. In the absence of such approaches, we can instead compare SIMRT to processes that are currently used in general regression testing applications, that might likewise be employed in our setting.

A second important issue regarding SIMRT involves whether either or both of its component techniques, selection and prioritization, play a role in its cost-effectiveness (or lack thereof), and if so, to what extent. Understanding this issue is important to any efforts to extend the approach further.

We thus designed a study focusing on three research questions:

RQ1: How do the *efficiency* and *effectiveness* of SIMRT, considering only its regression test selection component, compare to those of the retest-all technique and a state-of-the-art RTS technique.

RQ2: How does the *effectiveness* of the TCP technique employed by SIMRT compare to that of random test case orders when just the prioritized selected test cases are considered?

RQ3: How does the *effectiveness* of the TCP technique employed by SIMRT compare to that of traditional additional-block-coverage prioritization and random test case orders when the entire prioritized test suite is considered?

RQ1 lets us consider the overall efficiency and effectiveness of SIMRT focusing on its regression test selection component, compared to the baseline approach in which no selection is performed and to a state-of-the-art RTS technique. RQ2 lets us consider whether prioritization of just those test cases selected by SIMRT helps detect races more quickly than leaving them unprioritized. RQ3 lets us consider the overall effectiveness of prioritization if complete test suites are used.

4.1 Objects of Analysis

We chose nine open source Java objects, which are representative of real-world code and have been widely used in academic research. These included one object downloaded from the Software-artifact Infrastructure Repository (SIR) [5], five objects from JDK, and three larger objects [18, 25, 49]. The objects include both closed code units (code units equipped with test drivers) and open code units (libraries that require test drivers to close them).

Among the closed objects, WEBLECH is a multi-threaded web site download and mirror tool; it accepts both command line options and a configuration file that specifies settings such as URL, search options (e.g., BFS, DFS), and number of threads. JIGSAW is a leading-edge Web server platform; it accepts configuration files on both client and server sites, and multiple main entry points with command line options.

Among the open objects, LANG is part of the Apache Common Lang project. LOG4J is a Java logging application. HASHMAP, TREEMAP, ARRAYLIST, HASHTABLE, and BITSET are synchronized collection classes provided by Sun Microsystems' JDK. To close these objects, we wrote test drivers that utilize unit test cases generated by RANDOOP (described later). Because RANDOOP does not support multi-threaded programs, the test drivers also create sets of threads that concurrently execute the methods.

We utilized two versions of each of the nine objects. Table 1 lists our object versions along with some of their characteristics, including the number of lines of non-comment code (NLOC, column 2) and the number of shared variables identified in the modified versions (SVs, column 6) using the technique described in Section 3 (the numbers in parentheses indicate the number of impacted shared variables). Other columns are described later.

To address our questions, we also required multi-threaded test cases. Our objects, however, were either not equipped with such test cases, or the test suites supplied with them were too small (e.g.,

Table 1: Objects of Analysis and their Characteristics

Program	NLOC	|T|	PRs	RRs	SVs (ISVs)	ISV_{cov}s
LANG (V)	993	29104	-	-	-	-
LANG (V')	990		8	8	19 (6)	6
HASHMAP (V1.2)	852	26405	-	-	-	-
HASHMAP (V1.59)	1,014	-	59	10	230 (61)	51
TREEMAP (V1.2)	1,321	28203	-	-	-	-
TREEMAP (V1.56)	1,384	-	16	0	458 (17)	12
ARRAYLIST(V1.2)	727	34202	-	-	-	-
ARRAYLIST (V1.43)	747	-	22	3	352 (33)	24
HASHTABLE (V1.2)	3,041	28757	-	-	-	-
HASHTABLE (V1.55)	4,111	-	3	2	412 (36)	28
BITSET (V1.2)	194	30500	-	-	-	-
BITSET (V1.55)	486	-	4	2	240 (13)	10
LOG4J(V1.2.13)	15,331	32065	-	-	-	-
LOG4J (V1.2.8)	15,366	-	6	2	171 (63)	44
WEBLECH(V0.02)	16,393	9601	-	-	-	-
WEBLECH (V0.03)	16,633	-	29	1	23 (14)	12
JIGSAW(V2.2.0)	90,331	18805	-	-	-	-
JIGSAW (V2.2.6)	101,207	-	69	3	3452 (488)	209

fewer than 50 test cases) to allow SIMRT to operate as intended. To simulate a resource-constrained testing environment in which it makes sense to utilize approaches such as SIMRT, we chose to create test cases using a testing time budget, that limits the maximum number of test cases that can be executed for a given object. The testing time budgets we chose were selected to be relevant to the size and complexity of the objects. For the six smaller and less complex objects, we chose a testing budget of 12 hours (i.e., testing that can be performed overnight). For the three larger and more complex objects, we chose a testing budget of 60 hours (i.e., testing that can be completed over a weekend.) Both of these are practically realistic choices that engineers could make given that the testing required by our approach can be conducted automatically without the need for human intervention, yet must (like any validation effort) be conducted within some fixed time period. Note that the testing time we measured is based on the time required to run the instrumented original object with race detectors in place; thus, the numbers of generated test cases depend on the instrumentation overhead for different object objects.

The test cases we used were created using test case generation techniques relevant to the objects. Because our goal is to detect races, a test case must include two components: test input data and specified thread interleavings [41]. For objects WEBLECH and JIGSAW, which accept configuration files, we used incremental covering arrays [12] to generate test inputs. They also accept inputs via command line options, and such inputs were randomly generated. For the closed objects, which were not equipped with test drivers, we first used RANDOOP [31] to generate unit test cases for each method, and then we created multiple threads to concurrently execute these methods. Note that the number of threads is another input argument. For all object objects, the number of threads (if mutable) associated with each test input was randomly generated in a range from 2 to 100. Column 3 of Table 1 (|T|) lists the numbers of test cases ultimately utilized for each object.

To address our research questions we also needed to know what races (if any) existed in our object objects. To determine this, we ran all test cases on the original version of each object using the modified version of RACEFUZZER (described in Section 2). We discovered that the original object versions contained many potential and real races. In this work we are interested only in regression races (i.e., races introduced by code modifications); not residual races (races that persist across versions of the modified objects). Hence, we located the causes of such races in the original object versions, and corrected the code in both versions to ensure that they would not occur in either. Next, we executed all of the test cases associated with each original object on its modified version with the race detector and verifier enabled. This yielded a set of regression races that had been introduced by the code changes made to the modified versions.

Because non-determinism may cause race detectors to report different results on different object executions, we repeated the foregoing process ten times for each object pair, and accumulated any newly detected races. (The fluctuation in numbers of races reported on different runs was actually quite small, and is discussed further in Section 5.) Columns 4 and 5 of Table 1 list the numbers of potential and real regression races (PRs and RRs) discovered in the foregoing process. Column 7 lists the number of impacted variables (ISV_{cov}s) that are covered in the modified objects.

4.2 Variables and Measures

4.2.1 Independent Variable

Our independent variable involves the techniques used in our study. As noted earlier, we consider SIMRT, the traditional retest-all technique (RTA), and the traditional DEJAVU RTS technique (hereafter referred to as RTSC). With all three of these, we enable the race detector and verifier. We use SIMRT_S to denote SIMRT with just the RTS technique enabled.

We refer to SIMRT when it also employs a TCP technique as SIMRT_{SP}. When considering RQ2 and just selected test cases, we compare SIMRT_{SP} to a random test case ordering applied to just the selected test cases (denoted here by RANDOM_{SP}).

When considering RQ3, in which all test cases are prioritized, we compare SIMRT (denoting this version of the technique by SIMRT_{AP}) to two approaches: a random test case ordering applied to all test cases (denoted here by RANDOM_{AP}) and an application of the *additional block coverage* prioritization technique (denoted here by ABC_{AP}) described in Section 2.

4.2.2 Dependent Variables

As dependent variables, we chose metrics allowing us to answer each of our three research questions.

Regression test selection. To measure the *efficiency* of regression test selection we measured *testing time* by adding relevant measures, including the time required for escape and impact analyses (if any), the time required for running regression test selection algorithms (if any) and the time required to execute test cases.[1] To obtain these times, we applied RTSC and SIMRT_S and measured relevant analysis and test selection times. Then, to account for possible differences in testing times per execution of sets of test cases, we executed each object ten times on the various sets of selected test cases (or in the case of RTA, on all test cases), and calculated the average time required to execute the test cases.

To measure race detection *effectiveness*, we compare the numbers of potential and real races detected by test cases selected by RTA, RTSC, and SIMRT_S.

Test case prioritization. To measure the *effectiveness* of test case prioritization we considered the rate at which test cases detect both potential and real races. To measure such rates, we use the well-known metric $APFD$ (Average Percentage Fault Detected) [37]. Let T be a test suite containing n test cases, let T_p be a prioritized version of T, and let R be a set of m races revealed by T. Let TR_i

[1]When measuring testing time, we do not include time spent on activities performed in the preliminary phase of regression testing (prior to the time at which the modified object is available for testing, and when testing time becomes a critical issue); these include collection of test history information and construction of control flow graphs for the original object.

be the first test case in ordering T_p of T that reveals race i. The $APFD$ for test suite T_p is given by the equation:

$$APFD = \frac{\sum_{i=1}^{|m-1|} |TR_i|}{n*m} + \frac{1}{2n}$$

$APFD$ ranges from 0 to 100, with higher values indicating faster rates of race detection.

The $APFD$ metric can be applied to orderings of the entire test suite T, or to orderings of just the set of selected test cases T'. In the former case, $n = |T|$, and in the latter case, $n = |T'|$.

4.3 Study Operation

We conducted our experiment runs on a Linux cluster with 1440 AMD cores housed in 42 nodes with 128GB per node. We used a modified version of RACEFUZZER for race detection and verification (see Section 2). Among our objects, the instrumentation overhead incurred by our RACEFUZZER ranged from 1.5X to 20X. The remaining components in our framework (i.e., *ImpAnalyzer*, *Matcher*, *Selector* and *Ranker*) were implemented using Java by following the algorithms described in Section 3.

As noted in [4], executing a program once per input is sufficient to detect most, if not all, of the concurrent faults detectable under that input, because progams tend to follow the same interleaving patterns during different executions. As we discuss in Section 5, executing a program on the same input additional times may occasionally uncover additional races, but the benefit of doing this may be outweighed by the increased cost of testing. Therefore, the data reported in this study for each technique was obtained by executing each object once. However, we discuss the impact of race detection effectiveness given multiple test runs in Section 5.

4.4 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our objects and test cases. Other objects and test cases may exhibit different behaviors and cost-benefit tradeoffs. However, we do reduce this threat to some extent by using several varieties of well studied open source code objects for our study, and test suites generated by practical approaches.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

Where construct validity is concerned, our measurements of efficiency of regression test selection focus on the time required for analysis and test execution. However, other costs such as test setup and maintenance costs can play a role in technique efficiency. Our measurement of effectiveness for test case prioritization is $APFD$. Although $APFD$ does have certain limitations [6], it does provide a simple, intuitive measurement for rapid race detection.

4.5 Results and Analysis

4.5.1 RQ1: Regression Test Selection

Efficiency. Figures 5 and 6 show technique execution times in minutes for each of the three techniques and nine objects. For the six small objects using overnight testing, RTSC required less time than RTA, with savings ranging from 30.8% to 96.9%, and an average savings of 32.3% across all six objects. SIMRT achieved even greater savings than RTA on the smaller objects, with an average savings of 89.9%, and savings on individual objects ranging from 46.2% to 99.9%. For the three objects using over-the-weekend testing, RTSC required less time than RTA on only one object (LOG4J)

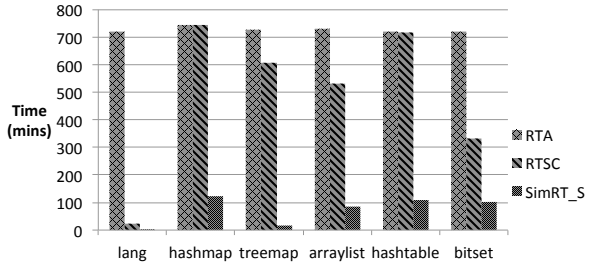


Figure 5: Efficiency: testing times for smaller objects

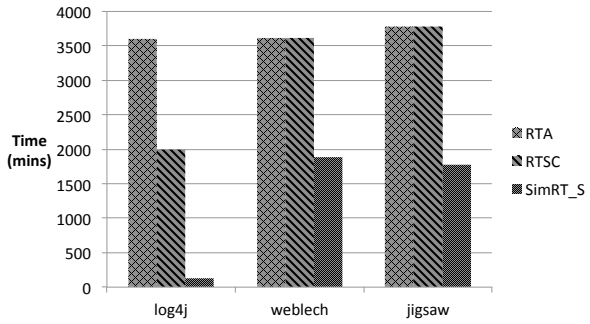


Figure 6: Efficiency: testing times for larger objects

with a savings of 44.8%. SIMRT achieved greater savings than RTA on all three objects, with savings of 96.5% on LOG4J, 47.9% on WEBLECH, and 52.9% on JIGSAW.

We also measured the time required by RTSC and SIMRT to perform analysis tasks. Due to space limitations we do not present all of the data, but overall, while SIMRT required greater analysis times than RTSC, its analysis times never exceeded 45 seconds, which accounted for less than 0.5% of technique runtime overall. The time spent on escape analysis and impact analysis never exceeded 308 seconds. We also measured the time spent by SIMRT on race verification, and the results ranged from 7% to 11.9% of total technique runtime. The remaining time was spent on analysis tasks and race detection. Verification time varied with the number of detected potential races and the number of test runs needed to verify each potential race.

Testing times did vary across objects. The number of test cases selected by techniques depends on the program locations in which changes occur or in which shared variables are impacted, and this, in turn, affects testing time. For example, in WEBLECH, many changes occur inside the `main` function, causing all test cases to be selected by RTSC; thus, in this case, no time is saved.

Effectiveness. We consider whether races (both potential and real) detected by RTA and RTSC can also be detected by SIMRT. Table 2 shows the numbers of potential and real races detected for each of the nine objects by RTA, RTSC and SIMRT. Because the data reported for each technique is based on a single run, the numbers of races detected by RTA do not necessarily equal the numbers in Table 1; negative numbers in parentheses indicate the numbers of races missed compared to those known to be detectable.

For all nine objects, RTSC detected 100% of the potential and real races detected by RTA. SIMRT, on the other hand, detected 100% of the potential races on seven of nine objects and 100% of the real races on all nine. On LOG4J and JIGSAW SIMRT missed one and two potential races, respectively. This sacrifice in effectiveness is relatively small, with only 1.3% of the potential races detected by RTA and RTSC missed across the nine objects. Meanwhile, the associated savings in terms of testing time was large.

Table 3: APFD Values for Selected and All Test Cases

Prog.	*Potential Races*					*Real Races*				
	SimRT$_{SP}$	Random$_{SP}$	SimRT$_{AP}$	ABC$_{AP}$	Random$_{AP}$	SimRT$_{SP}$	Random$_{SP}$	SimRT$_{AP}$	ABC$_{AP}$	Random$_{AP}$
LANG	96.9	50.3	100.0	71.5	52.2	96.9	50.3	100.0	71.5	52.2
HASHMAP	94.2	58.9	95.7	57.4	59.1	96.8	66.2	97.2	58.8	60.0
TREEMAP	98.1	39.8	100.0	48.5	40.8	-	-	-	-	-
ARRAYLIST	89.1	54.8	91.4	70.6	48.4	92.3	64.5	93.8	72.9	51.4
HASHTABLE	92.2	42.4	94.2	58.9	54.2	92.2	42.4	94.3	58.9	54.8
BITSET	97.5	64.6	96.8	88.4	69.2	97.8	70.4	96.8	88.6	71.5
LOG4J	85.7	47.5	90.0	47.8	48.5	89.2	56.5	90.6	48.4	48.8
WEBLECH	100.0	99.8	100.0	83.2	64.4	100.0	99.8	100.0	83.4	64.5
JIGSAW	88.4	52.8	91.2	72.9	56.6	93.6	58.6	94.8	78.0	61.4

Table 2: Race Detection Effectiveness

Prog.	*Potential races*			*Real races*		
	RTA	RTSC	SimRT	RTA	RTSC	SimRT
LANG	8	8	8	8	8	8
HASHMAP	59	59	58 (-1)	10	10	10
TREEMAP	16	16	16	0	0	0
ARRAYLIST	22	22	22	3	3	3
HASHTABLE	3	3	3	2	2	2
BITSET	4	4	4	2	2	2
LOG4J	6	6	6	2	2	2
WEBLECH	29	29	29	1	1	1
JIGSAW	68(-1)	68(-1)	66(-3)	3	3	3

4.5.2 RQ2 and RQ3: Test Case Prioritization

Table 3 summarizes the APFD values computed for the test suites and test case orders utilized in our study across all nine objects, for both potential and real races, with $SimRT_{SP}$ and $Random_{SP}$ applied to selected test cases, and with $SimRT_{AP}$, ABC_{AP} and $Random_{AP}$ applied to the entire test suites.

To compare the effectiveness of different techniques, we display results in terms of *improvement in race detection rate*, calculated as: $\frac{APFD_{T1}-APFD_{T2}}{APFD_{T2}}$*100%. This indicates the percentage improvement in APFD achieved by technique $T1$ over technique $T2$. For example, for potential race detection, on object LANG, the improvement in APFD achieved by $SimRT_{SP}$ over $Random_{SP}$ was $\frac{96.9-50.3}{50.3}$*100% = 92.6%.

RQ2: Prioritization Results on Selected Test Cases.

On the six small objects, SIMRT$_{SP}$ outperformed RANDOM$_{SP}$ in terms of APFD. The improvement ranged from 50.9% to 146.5% for potential races, and from 38.9% to 117.5% for real races, and the average improvement across the five objects was 88.3% for potential races and 67.7% for real races. On the three large objects, SIMRT$_{SP}$ outperformed RANDOM$_{SP}$ in terms of rate of race detection, with improvements of 80.4%, 0.2% and 67.4% for potential races, and 57.9%, 0.2% and 59.7% for real races.

On WEBLECH, RANDOM$_{SP}$ performed almost as effectively as SIMRT$_{SP}$ because in that case, $ISVP$s were concentrated in several methods, making it easier to achieve high coverage per selected test case, and difficult to achieve additional coverage.

RQ3: Prioritization Results on Entire Test Suites.

On the six small objects, when compared to ABC$_{AP}$, SIMRT$_{AP}$, improved the potential race detection rate by 9.5% to 106.2%, with an average improvement of 52.0%. When compared to RANDOM$_{AP}$, SIMRT$_{AP}$ improved the potential race detection rate by between 39.9% and 145.1%, with an average of 83.5%. Where detection of actual races is concerned, when compared to ABC$_{AP}$, SIMRT$_{AP}$ improved the detection rate by 9.3% to 65.3%, with an average of 40.7%. When compared to RANDOM$_{AP}$, SIMRT$_{AP}$ improved the detection rate by 35.4% to 91.6%, with an average of 68.7%.

On the three large objects, when compared to ABC$_{AP}$, SIMRT$_{AP}$ improved the potential race detection rate by 88.3%, 20.2% and 25.1%, respectively. When compared to RANDOM$_{AP}$, SIMRT$_{AP}$ improved the potential race detection rate by 85.6%, 55.2% and 61.1%, respectively. Where detection of real races is concerned, when compared to ABC$_{AP}$, SIMRT$_{AP}$ improved the race detection rate by 87.2%, 19.9% and 21.5%, respectively. When compared to RANDOM$_{AP}$, SIMRT$_{AP}$ improved the race detection rate by 85.7%, 55.0% and 54.4%, respectively.

5. SUMMARY AND DISCUSSION

Summary of Results.

SIMRT's RTS component was substantially more efficient in terms of testing time than the retest-all and traditional RTS techniques. Meanwhile, SIMRT detected the same sets of real races as the baseline techniques. Although a few potential races were missed, the number was small.

SIMRT's TCP component, employed on selected test cases, was substantially more effective in terms of $APFD$ than the random test case ordering. When employed on entire test suites, the TCP component was substantially more effective in terms of $APFD$ than both additional-block-coverage and random techniques. Meanwhile, in this case, SIMRT was able to detect potential races that were not detectable when operating on a subset of the test cases.

If these results generalize to other real objects and RTS and TCP techniques, then *if engineers wish to target race detection,* SIMRT *is the best technique to utilize.*

We now explore additional observations relevant to our study.

Multiple Runs. In our study, we executed each object once under the selected test cases for each technique to assess race detection effectiveness; a practical approach in resource-limited testing environments. Although empirical studies have shown that the fluctuation in race detection among multiple runs does not exceed 0.7% [4], we wished to determine whether multiple test runs could impact the effectiveness of the techniques studied.

To do this, for each RTS technique, we ran each modified object with the potential race detector and verifier ten times. The results indicated no differences in real race detection across the ten runs, and differences in potential race detection were revealed in only two cases. Specifically, on HASHMAP, SIMRT detected one extra race in two of the ten test suite runs, and missed one race in one test suite run. RTA and RTSC did not display any differences. On JIGSAW, RTA detected one extra race in two of the ten runs and missed one race in one run. RTSC detected one extra race in three of the ten runs, and SIMRT detected two extra races in three of the ten runs, missing one race in one run. These results imply that conducting one run for each test case is likely to be a reasonable level of effort in resource-limited testing environments.

Effects of Recording Thread Coverage. SIMRT selects test cases that potentially cover impacted shared variable pairs in the modified object with different threads. As such, SIMRT records thread IDs for test history construction. To determine whether this is useful, we further investigated whether savings could be attained by considering thread coverage. We considered percentages of test cases selected without using thread coverage, labeling this ap-

proach SIMRT$_N$. SIMRT$_N$ selects test cases that potentially cover impacted shared variable pairs in the modified object, regardless of the number of threads involved in covering each pair.

For all nine objects, SIMRT yielded savings over SIMRT$_N$; these ranged from 16.5% to 83.7% with an average of 77.4% for the six smaller objects and from 34.4% to 56.6% with an average of 44.9%, for the three larger objects. Therefore, the approach substantially improved the efficiency of regression test selection.

Numbers of Selected Test Cases. The numbers of test cases selected by various techniques is another metric for measuring the *efficiency* of regression test selection, providing an implementation-independent view of efficiency. For the nine objects, RTSC selected smaller test suites than RTA in many cases, with overall selection percentages ranging from 45.8% to 100%. On four of the nine objects, however, RTSC selected more than 90% of the test cases, and in two cases it selected 100% of the test cases. SIMRT, on the other hand, pruned away larger portions of the test suites, with selection percentages ranging from 0.1% to 51.8%. In fact, for seven objects, SIMRT selected fewer than 15% of the test cases.

Exposing Other Faults. SIMRT specifically targets races, and therefore, may not be effective at detecting other classes of faults that RTA or RTSC can reveal. As such, it is worth investigating the performance of RTSC when combined with SIMRT. We thus compared the testing time required by RTSC (with instrumentation for race detection) to that required by the use of both SIMRT and RTSC without instrumentation (denoted by RTSC$_N$). Both RTSC and RTSC$_N$ detect races and other output-based regression faults. This comparison showed that the savings of RTSC$_N$ over RTSC ranged from 43.5% and 67.6% (average 57.36%) across the six smaller objects. On the larger objects, the savings were 68.5%, 22.9% and 31.2%, respectively. This suggests that race detection and traditional fault detection should be considered separately.

Results such as these should be qualified. If SIMRT does not substantially reduce the number of selected test cases over RTSC, RTSC$_N$ may not achieve savings. Further, if engineers wish to detect both data races and other regression faults, and SIMRT substantially reduces the number of test cases selected over traditional RTS techniques, the best approach to use is: 1) run test cases selected by traditional RTS techniques on uninstrumented objects and 2) run test cases selected by SIMRT on instrumented objects. Race detection overhead varies across different types of applications. For example, I/O-intensive applications tend to have small race detection overhead, and hence would benefit less from SIMRT.

Further Discussion. We have used SIMRT to target data races, but it can be adapted to detect other types of concurrency faults such as atomicity violations [32, 33] and deadlocks [1, 19] by adjusting the contents of coverage targets. For example, to detect atomicity violations, instead of identifying potential impacted shared variable pairs as coverage targets, SIMRT can identify potential *unserializable interleavings* [33] as coverage targets.

As Table 1 shows, for some objects, there were a few uncovered impacted shared variables remaining in the modified versions after running the existing test cases. These variables may also cause data races. Covering the impacted shared variables pairs associated with these variables requires additional test cases or possibly thread interleavings. To address this, we intend to further explore extending SIMRT using test suite augmentation techniques [39].

6. RELATED WORK

There has been a great deal of work on analyzing, detecting, and testing for data races [9, 34, 40, 41, 43]; however, existing techniques do not consider software evolution.

There has been some work on selecting and prioritizing schedules for testing multithreaded programs such that faults can be exposed faster [3, 26]. For example, CHESS prioritizes the exploration of program state spaces to detect potentially erroneous interleavings. Again, these techniques do not consider code changes.

There have been several techniques presented for systematically exploring schedules in multi-threaded programs across versions [13, 17]. Gligoric et al. [13] reuse results from exploration of one program version to speed up exploration of the next program version. Jagannath et al. [17] use information about program changes in software evolution to prioritize the exploration of schedules. These techniques, however, target exploration of schedules within individual test cases and do not address the challenges of regression testing involving large sets of test cases. That said, we believe these techniques can also be utilized by SIMRT. For example, once test inputs have been selected or prioritized by SIMRT, we can further apply these techniques to select or prioritize thread schedules for each of the test cases.

Recent work by Deng et al. [4] studies how existing concurrency fault detection tools work for a set of test inputs. They propose a technique that first measures coverage of a program, and then selects a subset of test inputs to test for data races and atomicity violations on that program. This technique focuses, however, on single version programs and does not consider code changes. In contrast, we reuse coverage information on the old programs and select test inputs to test the new programs.

There has been a great deal of research on improving regression testing through regression test selection and test case prioritization (e.g., [7, 8, 21, 24, 30, 36, 37, 46, 51, 53]) – Yoo and Harman [52] provide a recent survey. Here, we restrict our attention to techniques that share similarities with ours.

Some RTS techniques target specific fault classes [23, 50, 55]. Our own previous work [55] selects test cases associated with system changes, but only those that are relevant to a set of internal oracles that are known to be important for the system under test. Staats et al. [47] propose a TCP technique that favors test orderings in which many variables impact the test oracle's result early in test execution. However, all the foregoing techniques focus on sequential programs, and do not address concurrency faults.

7. CONCLUSION

We have presented an automated regression testing framework, SIMRT, for use in detecting races that are induced due to code changes. SIMRT employs both RTS and TCP techniques. We have conducted an empirical study applying SIMRT to nine concurrent code objects. We have empirically compared SIMRT to traditional baseline techniques, and our results suggest that SIMRT's test selection component is more effective and efficient than other approaches in terms of race detection. Our results also show that the TCP technique employed by SIMRT is more effective than a traditional TCP technique and a random ordering in terms of rate of race detection.

In the future, we intend to perform more extensive empirical studies to evaluate SIMRT and extend SIMRT to detect other types of concurrency faults such as atomicity violations and deadlock. Finally, we will investigate the use of other regression testing techniques such as test suite augmentation in conjunction with SIMRT.

8. ACKNOWLEDGMENTS

This work has been supported in part by the Air Force Office of Scientific Research through award FA9550-10-1-0406 and the Army Research Office through award W911NF-13-1-0154.

9. REFERENCES

[1] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 Workshop on Parallel and Distributed systems: Testing and Debugging*, 2006.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[3] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[4] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2013.

[5] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[6] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), Sept. 2006.

[7] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), Sept. 2006.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), Feb. 2002.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th International Conference on Symposium on Operating System Design and Implementation*, 2000.

[10] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating systems Design and Implementation*, 2010.

[11] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[12] S. Fouché, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007.

[13] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, 2010.

[14] Guarded Blocks. http://docs.oracle.com/javase/tutorial /essential/concurrency/guardmeth.html.

[15] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[16] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[17] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[18] http://jigsaw.w3.org.

[19] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.

[21] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *IEEE International Conference on Software Maintenance*, 2008.

[22] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles*, 2011.

[23] H. K. N. Leung and L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance: Research and Practice*, 2(4), 1990.

[24] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4), Apr. 2007.

[25] http://logging.apache.org/log4.

[26] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[27] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[29] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[30] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.

[31] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, 2007.

[32] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[33] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for*

Programming Languages and Operating Systems, 2009.

[34] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[35] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), Aug. 1996.

[36] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2), Apr. 1997.

[37] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), Oct. 2001.

[38] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2001.

[39] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[41] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[42] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009.

[43] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[44] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[45] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[46] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.

[47] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*, 2012.

[48] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007.

[49] http://weblech.sourceforge.net.

[50] L. White and B. Robinson. Industrial real-time regression testing and analysis using firewalls. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004.

[51] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.

[52] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification and Reliability*, 22(2), 2012.

[53] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009.

[54] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[55] T. Yu, , X. Qu, M. Acharya, and G. Rothermel. Oracle-based regression test selection. In *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation*, 2013.

[56] T. Yu, W. Srisa-an, and G. Rothermel. Simtester: A controllable and observable testing framework for embedded systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[57] T. Yu, W. Srisa-an, and G. Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *IEEE 24rd International Symposium on Software Reliability Engineering*, 2013.

[58] T. Yu, W. Srisa-an, and G. Rothermel. Simracer: An automated framework to support testing for process-level races. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[59] W. Zhang, C. Sun, and S. Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[60] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.