Testing Concurrent Programs to Achieve High Synchronization Coverage

Shin Hong[†], Jaemin Ahn[†], Sangmin Park[*], Moonzoo Kim[†], and Mary Jean Harrold[*]

[†]Computer Science Department
KAIST, South Korea
{hongshin|jaemin|moonzoo}@cs.kaist.ac.kr

[*]College of Computing
Georgia Institute of Technology, Atlanta, GA
{sangminp|harrold}@cc.gatech.edu

ABSTRACT

The effectiveness of software testing is often assessed by measuring coverage of some aspect of the software, such as its code. There is much research aimed at increasing code coverage of sequential software. However, there has been little research on increasing coverage for concurrent software. This paper presents a new technique that aims to achieve high coverage of concurrent programs by generating thread schedules to cover uncovered coverage requirements. Our technique first estimates synchronization-pair coverage requirements, and then generates thread schedules that are likely to cover uncovered coverage requirements. This paper also presents a description of a prototype tool that we implemented in Java, and the results of a set of studies we performed using the tool on a several open-source programs. The results show that, for our subject programs, our technique achieves higher coverage faster than random testing techniques; the estimation-based heuristic contributes substantially to the effectiveness of our technique.

Categories and Subject Descriptors

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

General Terms

Algorithms, reliability

Keywords

Concurrency, thread interleaving, synchronization, coverage

1. INTRODUCTION

There has been much research that assesses the testing quality of sequential software by measuring coverage of the software based on structural coverage criteria, such as statement, branch, or data flow. Empirical studies show that there is a strong correlation between test suites with high coverage and the defect-detection ability of those test suites [4, 13, 15]. Some of these coverage criteria have been successfully implemented in tools that are used in industry.

Several coverage criteria have been presented for concurrent programs, such as synchronization-pair coverage and statement-pair coverage criteria [3, 19, 22]. The coverage criteria can be a metric for testing adequacy, but there has been little research for concurrent programs that aims at achieving high coverage, and thus investigating more program behaviors. Instead of directing testing to achieve more coverage, a common practice is stress testing—running a test with the same input repeatedly with the hope of executing different interleavings and finding bugs. A recent study shows that such stress testing can be unacceptably ineffective: such testing did not reveal a known concurrency bug even when executing the software for one week [14].

To execute a more diverse set of interleavings than stress testing for concurrent software, researchers have developed a random-testing approach, which inserts random delays at concurrent resource accesses [5, 18]. The underlying idea of these techniques is that injecting random delays perturbs the orderings of threads and results in diverse interleavings. The techniques are simple and scale to large systems, and thus, they have been used in production code, such as IBM WebSphere [1]. However, because of their random nature, the techniques may produce the same interleavings for many executions, and may not reveal some concurrency bugs that occur under specific interleaving [14].

To find concurrency bugs that occur under specific interleavings, researchers have developed bug-directed random testing techniques. The techniques target specific kinds of concurrency bugs, such as data race [16], atomicity violation [14], and deadlock [7]. The techniques predict possible concurrency bugs locations, and manipulate thread schedulers to guide the interleavings toward executing the predicted buggy locations. The main limitation of the techniques is that they are tailored to specific bug patterns, and may not explore diverse interleavings to reveal other kinds of faults. More systematic techniques have been proposed to avoid investigating the same interleavings repeatedly [12, 20]. Unlike random testing techniques, the systematic techniques are guaranteed to investigate different interleavings, and may reveal any kinds of concurrency bugs. However, the techniques are not scalable to large programs.

To address the limitations of existing techniques, we developed, and present in this paper, a novel thread-scheduling technique for achieving high test coverage for concurrent programs. Like traditional structural testing approaches for sequential programs, the goal of our technique is to achieve high coverage for concurrent program entities, such as statement pairs and synchronization pairs [19]. Specifically, our technique consists of *an estimation phase* and *a testing phase*. In the estimation phase, the technique dynamically builds a thread model of the target concurrent program, and, based on the model, estimates the coverage requirements. In the testing phase, the technique dynamically manipulates thread schedules to cover previously uncovered coverage requirements (computed in the estimation phase), and thus, reveals more diverse program behaviors.

Our technique has multiple benefits over existing techniques. First, unlike random testing techniques, our technique is based on measuring coverage against coverage criteria and covering more uncovered coverage requirements. Thus, running a program with our technique provides some statistics for determining the thoroughness of testing. Second, unlike bug-directed random testing techniques, our technique is not tailored to specific bugs, but is general enough to investigate diverse interleavings regardless of bug types. Thus, our technique can reveal any type of concurrency bug during testing. Third, unlike systematic techniques, our technique can scale to large real-world programs. We applied our technique to 20KLOC Java programs for our study, but we expect that we will be able to apply the technique to larger programs because of its simplicity.

We also describe the prototype implementation of the technique in Java, and present the results of several empirical studies that we performed to evaluate the *effectiveness* and *efficiency* of the technique over existing techniques. The first study investigates whether the technique achieves more coverage than other techniques. The results show that, for our subject programs, most often, our technique achieves higher coverage than random testing techniques given an execution limit. The second study investigates whether our technique achieves a given coverage limit faster than other techniques. The results show that our technique reaches higher coverage faster than other techniques. The third and fourth studies demonstrate the improved performance due to *the estimation-based heuristic*, which is a key asset of the technique (see Rule 3 of Algorithm 1). The third study shows that our technique estimates the coverage requirements almost precisely with respect to the actual execution of the requirements. The fourth study, which compares our technique with and without the estimation-based heuristic, demonstrates that the heuristic actually improves the overall performance of the technique.

The main contributions of the paper are

- The presentation of a new technique that aims at achieving high coverage faster for concurrent programs.
- A description of the implementation of the technique in Java; we made both our implementation and subject programs available for public use [2].
- A discussion of our empirical studies that show that, for our subject programs, our technique, to which the estimation-based heuristic contributes substantially, is more effective and efficient than existing techniques.

2. THREAD SCHEDULING TECHNIQUE

This section presents our thread-scheduling technique that manipulates thread schedules dynamically to increase test coverage for concurrent programs. Section 2.1 provides an overview of the technique, Section 2.2 presents definitions for understanding the technique, and Sections 2.3 and 2.4 describe, in detail, the two phases of the technique.

2.1 Overview

Our thread-scheduling technique consists of the two phases:

1. *Estimation phase* that identifies coverage requirements R
2. *Testing phase* that generates thread schedules to execute the coverage requirements in R

Figure 1 shows an overview of these two phases of the technique. In the Estimation phase, the technique executes a program P once with a test case to obtain a *thread model* $\mathcal{M}$ that consists of a set of thread executions. Then, based on the thread model, the technique estimates coverage requirements R that can be covered by possible thread-scheduling scenarios, and outputs R.

In the Testing phase, the technique inputs R, computed in the Estimation phase, and executes P with t using the scheduling controller to attempt to execute more coverage requirements in R. For each thread control point, the technique measures executed coverage requirements, marks the executed coverage requirements from R as covered, and generates the next thread schedule to cover uncovered coverage requirements in R.

2.2 Definitions

This section gives formal definitions that are used throughout the paper. Section 2.2.1 defines a formal thread model and its related functions, Section 2.2.2 defines an interleaved execution model on top of the thread model, and Section 2.2.3 defines coverage requirements and their satisfaction criteria.

2.2.1 Thread Model

We define a *thread model* $\mathcal{M}$ of a concurrent program as a finite set of threads, each of which consists of a finite sequence of atomic actions, where an action p has the following attributes:

- `thread`(p) is a thread executing p.
- `operator`(p) $\in$ `Sync` $\cup$ `Thread` $\cup$ `Data` indicates a type of p.
  - `Sync` = {`lock`, `unlock`}
  - `Thread` = {`thread-creation`, `thread-join`}
  - `Data` = {`read`, `write`}
- `operand`(p) indicates an operand of p.
  - For `operator`(p) $\in$ `Sync`, `operand`(p) is the corresponding lock.
  - For `operator`(p) $\in$ `Thread`, `operand`(p) is the corresponding thread.
  - For `operator`(p) $\in$ `Data`, `operand`(p) is the variable/memory location to read or write.
- `loc`(p) is the corresponding code location of p.
- `lockset`(p) is the set of locks held by `thread`(p) when p begins to execute.

We define the functions that relate two `lock` actions p and p' of the same thread (i.e., `operator`(p)=`operator`(p')=`lock` and `thread`(p)=`thread`(p')) as follows.

- `lockset`(p, p') is a set of locks that continuously guards p and p'.
- `next`(p) is the `lock` action of `thread`(p) that *first* accesses `operand`(p) after p.
- `prev`(p) is the `lock` action of `thread`(p) that *most recently* accessed `operand`(p) before p.[1]

[1]`next`(p) and `prev`(p) might be undefined when p is the last/first access of `operand`(p) by `thread`(p).

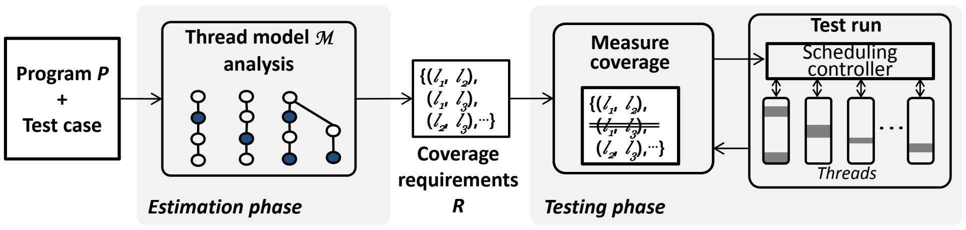
Program P
+
Test case
Thread model $\mathcal{M}$ analysis
{(l_1, l_2), (l_1, l_3), (l_2, l_3),···}
Coverage requirements R
Estimation phase
Measure coverage
{(l_1, l_2), ~~(l_1, l_3),~~ (l_2, l_3),···}
Test run
Scheduling controller
Threads
Testing phase

Figure 1: Overview of the thread-scheduling technique.

In addition, we define a *precedence relation* $\prec$ on the actions of $\mathcal{M}$ that represents ordering constraints between actions of two different threads t and t'. The ordering constraints are imposed at the time of thread creations.

> For any action p of thread t that occurs before t creates a new thread t' and for any action p' of t', $p \prec p'$.

2.2.2 Interleaved Execution Model

We define an *interleaved execution* σ of a concurrent target program as a sequence of actions of all the threads. During an interleaved execution σ, the program is at any program point with a state s. We introduce the following functions regarding σ and s:

- $\sigma[i]$ indicates ith action of σ.
- `enabled(`s`)` is a set of executable actions at s, through which s changes to another state s'.

For a given state s, each thread has at most one action in `enabled(`s`)` because a thread is a sequence of actions. A thread has no actions in `enabled(`s`)` if the thread is blocked because of synchronization.

2.2.3 Synchronization Coverage

Based on the thread model and the interleaved execution model, we define a coverage requirement for a synchronization-pair (SP) coverage metric. The definition is as follows:

Definition 1. **Synchronization-Pair (SP) Coverage Requirement**
A pair of code locations <l_1, l_2> is an SP requirement, if the following conditions hold for σ:

1. *l_1 and l_2 have lock statements that are executed on the same lock m (i.e., there is σ such that* `loc(`$\sigma[i]$`)=`l_1*,* `loc(`$\sigma[j]$`)=`l_2*,* `operator(`$\sigma[i]$`)=operator(`$\sigma[j]$`)=lock`*, and* `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)=`m *for some $i < j$).*
2. *There is no* `lock` *action on m between $\sigma[i]$ and $\sigma[j]$ (i.e., there is no k such that $i < k < j$,* `operand(`$\sigma[k]$`)= lock`*, and* `operand(`$\sigma[k]$`)=`m*).*

Note that this definition generalizes the original definition of Trainin et al. [19]. That definition does not consider a pair of code locations that are executed by the same thread (i.e., `thread(`$\sigma[i]$`)=thread(`$\sigma[j]$`)`) as an SP requirement, whereas our definition *does*. By doing this, we can express a larger set of coverage requirements, and thus, capture a greater number of interleaving executions.

Next, we discuss the conditions, under which SP requirements can be covered during execution. We formally define the satisfaction criterion as follows:

Definition 2. **SP Coverage Satisfaction Criteria**
For an execution σ of a program P and an SP requirement <l_1, l_2>, $\sigma \models < l_1, l_2 >$ if there exist i and j ($i < j$) such that

1. `loc(`$\sigma[i]$`)` $= l_1$ *and* `loc(`$\sigma[j]$`)` $= l_2$
2. `operator(`$\sigma[i]$`)=operator(`$\sigma[j]$`)=`*`lock`*
3. `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`
4. *there is no k such that $i < k < j$,* `operator(`$\sigma[k]$`)=lock`*, and* `operand(`$\sigma[k]$`)=operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`

2.3 Estimation Phase

The estimation phase computes and reports a set of SP requirements that can be satisfied by possible thread interleavings. To do this, the technique builds a thread model $\mathcal{M}$ by executing a program once and collecting data such as actions and threads. $\mathcal{M}$ has a set of executed threads, where each thread has a sequence of actions, and `lock` actions have their dynamic `lockset` information. From $\mathcal{M}$, the technique attempts to create every possible pair of `lock` statements. Then, the technique filters out some pairs that are definitely infeasible by checking (1) dynamic `lockset` relations (see AC2 and AC3 below), and (2) precedence relations (see AC4 below). The pairs that are not filtered out (i.e., accepted) are reported as the target SP requirements.

We formally define the *acceptance conditions (AC)* of the SP requirements as follows. For SP requirement <`loc(`p`)`, `loc(`q`)`> of `lock` actions p and q on the same lock m, the technique accepts the pair when the following conditions hold:

- If `thread(`p`)=thread(`q`)`,

(AC1) q=`next(`p`)`

- If `thread(`p`)`$\neq$`thread(`q`)`,

(AC2) `lockset(`p`, next(`p`))` $\cap$ `lockset(`q`)` $= \emptyset$
(AC3) `lockset(`p`)` $\cap$ `lockset(prev(`q`),` q`)` $= \emptyset$
(AC4) $q \nprec p$

AC1 accepts consecutive `lock` statements executed from the same thread as a feasible pair. AC2, AC3, and AC4 define the conditions for the pairs from different threads. AC2 implies that p, `next(`p`)`, and q should *not* be protected by a common lock, so that q can execute consecutively after p (i.e., before `next(`p`)`). For example, suppose there exists m such that $m \in$ `lockset(`p`, next(`p`))` $\cap$ `lockset(`q`)`. Then, p and `next(`p`)` are continuously protected by m, and thus, q cannot execute consecutively after p. Hence, <`loc(`p`)`,`loc(`q`)`> is filtered out. AC3 filters out infeasible conditions in a similar manner. AC4 filters out pairs that violates the precedence relation. If $q \prec p$, p cannot execute before q so that the corresponding pair is infeasible.

```
1a thread1() {                 1b thread2() {
2a   synchronized(m) {         2b   synchronized(m) {
3a   }                         3b   }
4a   synchronized(m) {         4b   synchronized(m) {
5a   }                         5b   }
6a }                           6b }
```

Figure 2: Example of a two threaded program.

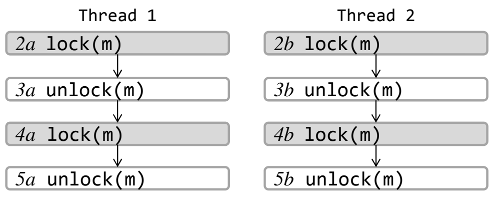

Figure 3: Thread model $\mathcal{M}$ of the target program.

To illustrate our technique, consider the simple Java program shown in Figure 2. The program has two threads, where each thread has two `synchronized` blocks on the same lock m. `Synchronized` blocks can be represented as `lock` and `unlock` statements, so `2a`, `4a`, `2b`, and `4b` correspond to `lock` statements, and `3a`, `5a`, `3b`, and `5b` correspond to `unlock` statements.

In the Estimation phase, our technique executes the program and obtains a corresponding thread model $\mathcal{M}$, as shown in Figure 3. We use an action $2a$ to indicate an action that executes a statement at `2a`, use an action $3b$ for a statement at `3b`, and so on.

For this example, there can be 12 ($= 4 \times 3$) possible pairs (i.e., SP coverage requirements) because there are four `lock` statements in the program. After checking the feasibility of these 12 requirements with $\mathcal{M}$, however, two of them are eliminated by AC1. For example, <`4a`,`2a`> is infeasible because $2a \neq$ `next`$(4a)$ (i.e., <`4a`,`2a`> violates AC1). <`4b`,`2b`> is infeasible for the same reason. Thus, there are 10 SP requirements that the Testing phase will try to cover:

<`2a`,`4a`>, <`2a`,`2b`>, <`2a`,`4b`>, <`4a`,`2b`>, <`4a`,`4b`>,
<`2b`,`4b`>, <`2b`,`2a`>, <`2b`,`4a`>, <`4b`,`2a`>, <`4b`,`4a`>.

2.4 Testing Phase

The testing phase takes the SP requirements and executes the target program P with the scheduling controller to achieve more coverage during the executions of P. During program executions, the technique invokes the scheduling controller before each `lock` action. The controller pauses or resumes threads to cover uncovered SP requirements. Note that this approach does *not* change P's semantics because the scheduling controller can only delay an execution of an action.

The algorithm for the scheduling controller for one test execution is shown in Algorithm 1. The algorithm receives an initial state s_0, a set of the SP requirements that have been covered by previous test executions (*covered*), and a set of uncovered target SP requirements (*uncovered*). For the first execution, *covered* is $\emptyset$ and *uncovered* is the set of SP requirements obtained in the estimation phase. At the termination of the algorithm (i.e., one test execution), the algorithm passes updated *covered* and *uncovered* to the next run.

The algorithm repeats scheduling decisions to cover uncovered SP requirements until all threads terminate (lines 3–38). For that purpose, the algorithm pauses or resumes a thread by manipulating *paused*, which is a set of `lock` actions that are paused by the scheduling controller. Note that action p can be executed only if $p \notin paused$ (lines 31–37). Thus, the algorithm can pause a thread t by adding currently enabled `lock` action p of t to *paused* (see lines 5–14). Also, the algorithm can resume t later by removing p from *paused* (see lines 18–29).

Algorithm 1 controls `lock` actions to generate thread scheduling as follows. The algorithm adds an enabled `lock` action p to *paused* if there is $<l_1, l_2>$ which is uncovered and `loc`$(p)= l_1$ or l_2 (lines 7–8), because controlling the execution of p may cover $<l_1, l_2>$. In addition, if p uses the same lock that is used by another action $p' \in paused$ (lines 10–12), the algorithm inserts p in *paused* to prevent p from blocking p' whose execution is controlled by the algorithm.

```
   Input:
   $s_0$: an initial state
   covered: a set of covered SP requirements
   uncovered: a set of uncovered SP requirements
   Output: Updated covered and uncovered
1  $paused \leftarrow \emptyset$;
2  $s \leftarrow s_0$;
3  while enabled($s$) $\neq \emptyset$ do
4      $p \leftarrow$ an action in enabled($s$)$\setminus paused$;
5      if operator($p$) =lock then
6          // Decides if $p$ is added to paused
7          if $\exists$ <$x$,$y$> $\in$ uncovered: loc($p$) =$x$ or $y$ then
8              Adds $p$ to paused ;
9          else
10             if $\exists p' \in paused$: operand($p$) =operand($p'$)
               then
11                 Adds $p$ to paused ;
12             end
13         end
14     end
15     if $paused$ = enabled($s$) then
16         // Selects an action in paused to execute later
17         // Rule 1
18         if $\exists p' \in paused$: <loc_last($p'$), loc($p'$)>
           $\in$ uncovered then
19             Removes $p'$ from paused;
20         else
21             // Rule 2
22             if $\exists p', p'' \in paused$: <loc($p'$), loc($p''$)>
               $\in$ uncovered and operand($p'$) =operand($p''$)
               then
23                 Removes $p'$ from paused;
24             else
25                 // Rule 3: the estimation based heuristic
26                 $p' \leftarrow$ an action in {$p'' \in paused$ |
                   num_cov_req($p''$, uncovered) is minimal};
27                 Removes $p'$ from paused;
28             end
29         end
30     end
31     if $p \notin paused$ then
32         $s \leftarrow$execute($s, p$); // updates $s$ by executing $p$;
33         if operator($p$) = lock then
34             // Updates coverage
35             Moves <loc_last($p$), loc($p$)> from
               uncovered to covered;
36         end
37     end
38 end
```

Algorithm 1: Scheduling Controller Algorithm

When no action can be executed because all enabled actions at a state s belong to *paused* (line 15), the algorithm selects an action p' in *paused* that seems best for covering uncovered SP requirements (lines 18–29). The algorithm determines p' based on the following three rules in the order given:

Rule 1: p' whose execution can immediately satisfy uncovered <`loc_last(`p'`)`,`loc(`p'`)`> (lines 18–19)

Rule 2: p' such that <`loc(`p'`)`, `loc(`p''`)`> is uncovered where p' and p'' operate on the same lock (lines 22–23)

Rule 3: p' such that p' has the smallest number of relevant coverage requirements that are uncovered (lines 26–27).

For Rule 1, the algorithm searches for an action p' in *paused* such that (<`loc_last(`p'`)`,`loc(`p'`)`>) is not covered where `loc_last(`p'`)` indicates a code location corresponding to the most recent `lock` action on the lock of p'. Note that this pair can be satisfied immediately by executing p', because the `lock` statement at `loc_last(`p'`)` was already executed before p'. If *paused* has no such action applicable for Rule 1, for Rule 2, the algorithm searches for the two actions p' and p'' in *paused* such that p' and p'' operate on the same lock and executing p' and then p'' satisfies a uncovered coverage requirement. If *paused* has such p' and p'', the algorithm removes p' from *paused* to execute.

If Rule 1 and Rule 2 cannot be applied, Rule 3 searches for p' that has the smallest number of *relevant* uncovered requirements. A number of relevant coverage requirements in *uncovered* to p'' is defined as `num_cov_req(`p''`, `*uncovered*`)` that returns a number of coverage requirements $< x, y >\in$ *uncovered* such that `loc(`p''`)`$=x$ or y. A main idea of Rule 3 is that p'' in *paused* has more chances to increase coverage if *uncovered* has more relevant coverage requirements to p''. Thus, Rule 3 removes p' from *paused* that has least potential benefit to hold. This heuristic is called *the estimation based heuristic*, because it utilizes a set of estimated target requirements *uncovered* to select p'. Note that this heuristic improves the performance of the algorithm substantially as demonstrated in Section 3.5.

Recall the example in Figures 2 and 3. For the first test execution in the testing phase, Algorithm 1 starts with *covered* $= \emptyset$ and *uncovered* as the 10 requirements in Section 2.3. At the initial state $s = s_0$, *paused* $= \emptyset$ and `enabled(`s`)`$=\{2a, 2b\}$. We explain the corresponding operations of the algorithm step by step as follows:

Loop iteration 1: [$enabled(s)$=$\{2a, 2b\}$, $paused$=$\emptyset$]
The algorithm selects $2a$ as p and inserts $2a$ into *paused* because <`2a`,`2b`>$\in$ *uncovered* (lines 7–8). Consequently, as $2a \in$ *paused* (line 31), no action executes and a current state does not change.

Loop iteration 2: [$enabled(s)$=$\{2a, 2b\}$, $paused$=$\{2a\}$]
The algorithm selects $2b$ as p and inserts $2b$ into *paused* because <`2a`,`2b`>$\in$ *uncovered*. As *paused* $= \{2a, 2b\} =$ `enabled(s)` (line 15), the algorithm picks p' in *paused* by applying Rule 2, as Rule 1 cannot be applied because neither `loc_last(`$2a$`)` nor `loc_last(`$2b$`)` is defined (line 18). Rule 2 selects $p' = 2a$ and $p'' = 2b$ and <`loc(`$2a$`)`, `loc(`$2b$`)`>$\in$ *uncovered* (line 22). Thus, the algorithm removes $2a$ from *paused* (line 23). However, because $2b \in$ *paused*, no action executes and a current state does not change.

Loop iteration 3: [$enabled(s)$=$\{2a, 2b\}$, $paused$=$\{2b\}$]
The algorithm selects $2a$ as p and inserts $2a$ into *paused* because <`2a`,`2b`>$\in$ *uncovered*. As *paused* $= \{2a, 2b\} =$ `enabled(s)` (line 15), the algorithm picks p' in *paused* by applying Rule 2, as Rule 1 cannot be applied because neither `loc_last(`$2a$`)` nor `loc_last(`$2b$`)` is defined (line 18). Rule 2 selects $p' = 2a$ and $p'' = 2b$ and <`loc(`$2a$`)`, `loc(`$2b$`)`>$\in$ *uncovered* (line 22). Thus, the algorithm removes $2a$ from *paused* (line 23). Then, because $2a \notin$ *paused* (line 31), the algorithm executes $2a$ and updates the current state (line 32).

Loop iteration 4: [$enabled(s)$=$\{3a, 2b\}$, $paused$=$\{2b\}$]
The algorithm selects $3a$ as p. Because $3a$ is not a `lock` action, the algorithm does not execute lines 6–13. Then, the algorithm does not execute lines 16–29, because *paused* $\neq$`enabled(`s`)`). Finally, it executes $3a$ and updates the current state (line 32).

Loop iteration 5: [$enabled(s)$=$\{4a, 2b\}$, $paused$=$\{2b\}$]
$4a$ is selected as p and added to *paused* (line 8), as <`4a`, `2b`> $\in$ *uncovered* (line 7). Then, as *paused* = `enabled(s)`, Rule 1 removes $2b$ from *paused* because <`loc_last(`$2b$`)`, `loc(`$2b$`)`>=<`2a`,`2b`> $\in$ *uncovered*. Because $4a \in$ *paused*, no action executes and a current state does not change.

Loop iteration 6: [$enabled(s)$=$\{4a, 2b\}$, $paused$=$\{4a\}$]
The algorithm selects $2b$ as p and inserts $2b$ in *paused* (line 8) because <`4a`, `2b`> $\in$ *uncovered* (line 7). Then, as *paused* = `enabled(s)`, Rule 1 removes $2b$ from *paused* because <`loc_last(`$2b$`)`,`loc(`$2b$`)`>=<`2a`, `2b`> $\in$ *uncovered* (line 32). Since $2b \notin$ *paused*, $2b$ is executed and <`2a`,`2b`> is moved from *uncovered* to *covered* (lines 35).

As another example to show how Rule 3 operates, suppose that test executions continued and *covered*=$\{$<`2a`,`2b`>, <`2a`,`4a`>,<`4a`,`4b`>, <`2b`,`2a`>,<`2b`,`4a`>, <`2b`,`4b`>,<`4b`,`4a`>$\}$ and *uncovered* = $\{$<`2a`,`4b`>,<`4a`,`2b`>,<`4b`,`2a`>$\}$.

Loop iteration 1: [$enabled(s)$=$\{2a, 2b\}$, $paused$=$\emptyset$]
The algorithm selects $2a$ as p and inserts $2a$ into *paused* (lines 7–8) because <`2a`,`4b`>$\in$ *uncovered*. Consequently, as $2a \in$ *paused* (line 31), no action executes and a current state does not change.

Loop iteration 2: [$enabled(s)$=$\{2a, 2b\}$, $paused$=$\{2a\}$]
The algorithm selects $2b$ as p and inserts $2b$ into *paused* because <`4a`, `2b`>$\in$ *uncovered* (line 7–8). As *paused* $= \{2a, 2b\} =$ `enabled(s)` (line 15), the algorithm picks p' in *paused* by applying Rule 3. This is because Rule 1 cannot be applied because neither `loc_last(`$2a$`)` nor `loc_last(`$2b$`)` is defined (line 18). Rule 2 cannot be applied because <`2a`, `2b`> $\notin$ *uncovered* and also <`2b`, `2a`> $\notin$ *uncovered* (line 22). Thus, Rule 3 selects $2b$ (line 26–27) because $2b$ has one relevant coverage requirement in *uncovered* (<`4a`, `2b`>) whereas $2a$ has two relevant coverage requirements in *uncovered* (<`2a`, `4b`> and <`4b`, `2a`>). Then, as $2b \notin$ *paused* (line 31), the algorithm executes $2b$ and updates the current state (line 32).

3. EMPIRICAL STUDIES

To evaluate our technique, we implemented it in a prototype tool in Java, and performed several empirical studies with the tool on a number of Java subjects. Section 3.1 describes the experimental setup, Sections 3.2 to 3.5 present the four studies we performed, and Section 3.6 discusses the threats to validity of the studies.

3.1 Experimental Setup

This section discusses the setup for our studies: the implementation of our technique, the subjects, and the independent and dependent variables.

3.1.1 Implementation

We implemented our technique in Java on top of the CalFuzzer framework [6]. We modified both the instrumentation and the scheduling-controller modules of the framework, and created new modules for the two phases of our technique (described in Section 2). Our modifications to Calfuzzer and the new modules consist of approximately 1910 lines of code.

The module for the estimation phase instruments concurrent Java programs to build a thread model. To do this, the module inserts probes before every synchronization operation (e.g., `synchronized` statements), shared-data access, and thread-related operation. Then, we run the instrumented program once to build a thread model. Based on the model, the module generates a set of SP coverage requirements and stores it in a text file.

The module for the testing phase takes the text file containing the set of SP coverage requirements, and executes the instrumented program multiple times to achieve high SP coverage. The main part of the module is the modification of the scheduling controller of the CalFuzzer framework. The scheduling controller of the original framework attempts to cover a likely-buggy interleaving at one program execution, whereas our scheduling controller attempts to cover as many coverage pairs as possible. Thus, our scheduling controller must maintain more runtime information than the original one. To reduce runtime overhead, we maintain the coverage information in a separate thread, and control the scheduler concurrently. In the module, we also built a timeout checker that kills the execution of the program when the program reaches a timeout limit.

3.1.2 Subjects

Table 1 shows the subject programs we used for our studies. The first column shows the type of the subject program. `Java Library` is a set of classes extracted from the `java.util.Collection` package. `Java Server` is a set of open-source server programs. For each program, the second column gives the name of the program, the third column shows the size of the program in lines of code, and the fourth and fifth columns show the numbers of threads and synchronization statements in the program, respectively.

We created test cases for each subject program. For the `Java Library` programs, we created two test cases for each class, where the program name with suffix 1 has a test case that creates several threads, each of which calls only one method,[2] and the program name with a suffix 2 has a test case that creates small number of threads, each of which calls several methods. For example, the test case for `ArrayList1` creates shared objects of the `ArrayList` class, which are shared in 26 threads, whereas the test case for `ArrayList2` creates shared objects and uses 4 threads. For the `Java Server` programs, we analyzed the subjects and created test cases manually. `cache4j` is a multi-threaded cache for Java objects. We created a test case that concurrently adds and removes Java objects to and from a cache server, respectively. `pool` is an object-pooling API for Java programs. We created a test case that concurrently calls pooling functions. `VFS2` is a Java virtual filesystem framework. We created a test case that concurrently calls file-system operations (e.g., open, read, and write) to a virtual filesystem object.

[2]The five test cases with suffix 1 are taken from the examples in the CalFuzzer distribution.

Table 1: Subject programs used for the studies.

Type	Program	Lines of Code	# of Threads	# of Sync. stmts.
Java Library	ArrayList1	7712	26	69
	ArrayList2	7712	4	67
	HashSet1	9028	21	67
	HashSet2	9028	3	66
	HashTable1	11431	5	96
	HashTable2	11431	5	116
	LinkedList1	7375	26	67
	LinkedList2	7375	15	66
	TreeSet1	5669	21	69
	TreeSet2	5669	3	67
Java Server	cache4j	1922	10	128
	pool	5536	10	280
	VFS2	22981	6	116

We ran our empirical studies on a Linux platform with Intel Core2 Duo 3.0GHz CPU and 16GB of memory. We used Sun Java SE 1.6.0 on top of Fedora Linux 9 (linux kernel 2.6.27).

3.1.3 Variables

The main independent variable is the thread-schedule generation technique: (1) our thread-scheduling algorithm (`TSA`), (2) `TSA-`*h* which is `TSA` without the estimation based heuristic (Rule 3 of Algorithm 1), and (3) 15 varieties of the random testing technique. The random testing techniques [5,18] insert random delays at shared resource accesses and synchronization operations in concurrent programs. Because a previous study [9] shows that the random parameters significantly influence the effectiveness of the random testing techniques, we vary the parameters of the random testing. Overall, we use three random testing techniques: (1) `RND-y`, which inserts `yield()` synchronization keywords at the shared resource accesses and synchronization operations, (2) `RND-s10`, which inserts random delays up to 10 milliseconds with the `sleep()` synchronization keyword, and (3) `RND-s100`, which inserts random delays up to 100 milliseconds with the `sleep()` synchronization keyword. For each random testing techniques, we insert delays with random probabilities: 0.1, 0.2, 0.3, 0.4, and 0.5. For example, `RND-s10` with probability 0.1 inserts a random delay once in 10 times at shared memory accesses, and when inserting a delay, it calls the `sleep()` function with less than 10 milliseconds. Thus, the total number of random testing techniques is 15 (= 3 techniques $\times$ 5 probability values). We did not include the parameter of the `sleep()` function over 100 milliseconds and random probability over 0.5, because our pilot studies showed that the effectiveness of the random testing techniques decrease when we increase the values above those limits.

The main dependent variables include (1) the number of covered SP coverage requirements and (2) the execution time to attain a certain goal. Both measures imply different results. Achieving greater coverage of the SP coverage requirements implies that a given technique is more effective than the techniques to which it is compared, whereas achieving coverage faster implies that a given technique is more efficient techniques to which it is compared. We measure coverage of the SP coverage requirements in Study 1, and both coverage and time in Studies 2 and 4.

Another dependent variable is the number of feasible SP coverage requirements. Our technique estimates a set of such requirements in the estimation phase, and uses these requirements in the testing phase (see Section 2). We measure the number of the SP coverage requirements that are actually covered during the testing phase. We measure this dependent variable in Study 3.

3.2 Study 1: Effectiveness

The goal of the study is to investigate whether `TSA` achieves higher coverage than random testing techniques. To do this, using the implementation of our technique, we

- Ran the estimation phase on each subject and, for each subject, created a set of SP coverage requirements
- Using the sets of SP coverage requirements, ran `TSA` and each of the 15 random testing techniques 30 times
  - For each of the 30 runs, executed the program 500 times and recorded the accumulated number of covered SP coverage requirements for each execution
  - Computed the average for the accumulated covered SP coverage requirements over the 30 runs

Table 2 shows the results of this study. The first column shows the program name. The second to tenth columns show the accumulated number of covered SP coverage requirements for the random testing techniques. Within each column for the random techniques, AVG gives the average of all results, and MIN and MAX give the minimum and maximum of the results, respectively. Finally, the 11th column shows the accumulated number of covered SP coverage requirements for `TSA`. Note that, we did not perform the study for `RND-s10` on `cache4j` and `RND-s100` on `cache4j`, `pool`, and `VFS` because it took longer than the time limit of 12 hours that we set for each experiment (i.e., 500 testing executions). Thus, for these programs, the results are shown as "-" in the table. For example, for `ArrayList1` with `RND-y`, we ran 30 experiments for each five different random probability parameter, got averages for each parameter, and thus got a total of five results. The average of the five values is 138.6, the minimum is 137.5, and the maximum is 140.0. For `ArrayList1` with `TSA`, we ran 30 experiments, and got an average of the 30 experiments as 181.6.

Figure 4 shows the results of this study in graphical form. For each graph, the horizontal axis represents the 500 testing executions performed in the study, and the vertical axis represents the number of covered SP coverage requirements from the testing executions. Each graph shows the results of `TSA` and the averages of `RND-y`, `RND-s10`, and `RND-s100`. For example, consider the graph of `pool`. There are only three values because `RND-s100` was not studied for this subject. In the graph, `TSA` (solid line) achieves a greater coverage than AVGs of `RND-y` and `RND-s10`, and the value at 500th execution is 2959.0 as in Table 2. The values of AVGs of `RND-y` and `RND-s10` are 1569.2 and 1990.3, respectively, as seen both in Table 2 and Figure 4.

We make three observations from the results of the study. First, most of the time, `TSA` achieves SP requirements that are greater than or equal to the maximum results of random testing techniques as shown in Table 2. To determine whether the results are statistically significant, we applied the Wilcoxon test with $\alpha < 0.05$ to compare `TSA` to MAX of `RND-s100`, which performs best in random testing techniques, for all subject programs. If the p-value is less than 0.05, `TSA` is statistically more effective than MAX of `RND-s100`. If the p-value is greater than or equal to 0.05, there is no statistical difference between `TSA` and `RND-s100`. The Wilcoxon test shows that for most of the subjects (`ArrayList1`, `HashSet2`, `HashTable1`, `HashTable2`, `LinkedList1`, and `TreeSet2`), the p-value is less than 0.05, and `TSA` performs more effectively than random testing techniques, including `RND-s100`.

Second, among the three random testing techniques, `RND-s100` performs slightly better than others. For `RND-y` and `RND-s10`, `RND-s10` always performs better. The AVG column of `RND-s10` is on average 25.7% higher than that of `RND-y`. For `RND-s10` and `RND-s100`, `RND-s100` performs better than `RND-s10` except for `HashSet2` and `TreeSet2`. We believe that longer random delays provide more chance of different interleavings, and thus `RND-s100` outperforms others. However, `RND-s100` is not always guaranteed to perform better than others, and moreover, it is not scalable to large programs, such as `cache4j` and `VFS2`. In contrast, `TSA` is scalable to work on the large programs.

Third, the effectiveness in the same random testing technique changes depending on the parameters. For example, for `RND-s100`, the results of the minimum and the maximum columns vary from 0.4% to 38.8% depending on the subject programs. This observation confirms the conclusion of the previous study [9] that the random parameters significantly influence the effectiveness of the random testing techniques. In addition, because there is no guidance for choosing the best parameter for random testing techniques, a developer can use `TSA` in practice, instead of using random testing techniques with different parameters.

3.3 Study 2: Efficiency

The goal of the study is to investigate the efficiency of the technique compared to random testing techniques. To do this, we compared the time for the techniques to reach a saturation point as the limit for covering SP coverage requirements. *Saturation-based testing* [17] stops testing when the coverage increment rate is less than a set threshhold. The original technique uses linear regression, with r^2 coefficient. We use the coefficient value 0.1 with a window size of 120 seconds. To perform this study, we

- Ran the estimation phase on each subject and, for each subject, created a set of SP coverage requirements
- Using the sets of SP coverage requirements, ran `TSA` and each of the 15 random testing techniques 30 times
  - For each of the 30 runs, executed the program for 30 minutes and recorded the accumulated number of covered SP coverage requirements for each execution
  - Computed the saturation point for each run
  - Computed the average saturation point (i.e., a pair of the average time and the average SP coverage requirements of the saturation points) over 30 runs

Table 3 shows the results of this study. The first column shows the program name. The second to fourth columns show the number of covered SP coverage requirements and the time of `RND-y`, `RND-s10`, and `RND-s100`, when they reach the saturation points. We recorded the *best results* for each random technique regardless of parameters. The final column shows the number of covered SP coverage requirements and the time for `TSA`. For example, for `ArrayList1`, `RND-y` reaches saturation within 411.5 seconds and covers 129.5

Table 2: Results of Study 1 (Effectiveness).

Program	RND-y			RND-s10			RND-s100			TSA
	AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	
ArrayList1	138.6	137.5	140.0	150.9	125.8	175.5	156.2	130.8	181.2	181.6
ArrayList2	111.5	108.1	114.8	132.2	117.5	142.4	132.7	116.7	143.7	141.8
HashSet1	83.6	79.8	86.0	134.0	112.7	153.2	138.0	115.2	154.0	152.6
HashSet2	104.7	100.3	107.8	112.8	100.9	120.8	111.6	99.8	120.6	120.8
HashTable1	15.2	15.0	15.6	24.0	23.8	24.0	24.0	23.9	24.0	24.0
HashTable2	393.2	371.5	415.5	452.5	405.9	496.3	461.0	405.9	508.3	538.0
LinkedList1	126.5	124.7	130.4	150.0	126.1	174.8	156.3	130.7	181.4	181.5
LinkedList2	116.3	110.6	121.3	128.6	112.3	142.1	129.2	111.6	143.6	141.7
TreeSet1	85.4	83.5	88.5	133.6	111.0	153.1	137.7	115.5	154.0	152.4
TreeSet2	109.6	107.0	111.3	110.0	96.1	119.6	109.3	97.6	119.5	120.5
cache4j	198.2	197.8	199.0	-	-	-	-	-	-	202.2
pool	1462.9	1359.6	1541.1	1990.5	1878.8	2102.9	-	-	-	2959.0
VFS2	239.1	237.2	241.1	234.3	216.7	250.8	-	-	-	262.0

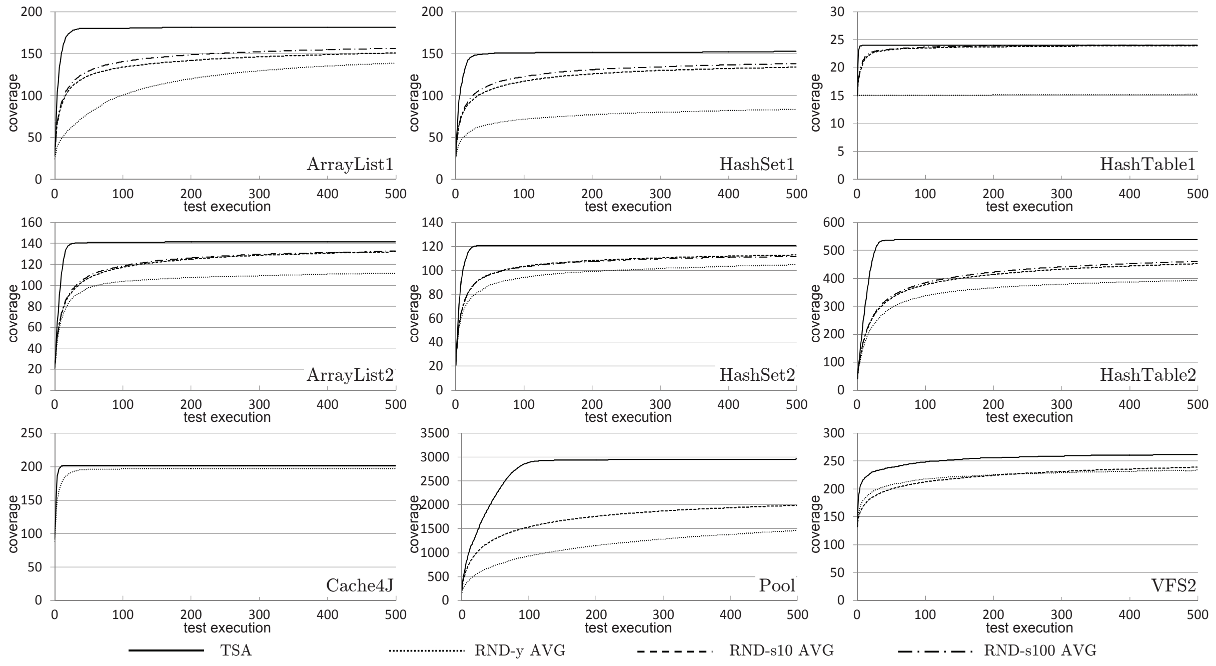

Figure 4: Results of Study 1 (Effectiveness): The number of covered SP requirements for 500 testing executions

SP coverage requirements, whereas TSA reaches saturation within 184.2 seconds and covers 181.2 SP coverage requirement. Note that some techniques do not reach a saturation point for some subjects in 30 minutes, which is the time limit we set for the experiment. In this case, we show the results as "-" in the table.

Figure 5 shows the results of this study graphically. The horizontal axis represents the execution time, and the vertical axis represents the number of SP coverage requirements. Each graph shows the results of TSA and the averages of RND-y, RND-s10, and RND-s100. Consider the graph of pool as an example. There are only two values because RND-s10 and RND-s100 did not reach a saturation point. In the graph, TSA (solid line) reaches a saturation point at about 431.4 second with 2950.5 coverage, whereas RND-y reaches a saturation point at about 388.5 second with 888.0 coverage.

The most important observation of this study is that TSA always reaches the saturation point faster and covers a greater number of SP coverage requirements than all random testing techniques. TSA reaches a saturation point in less than 200 seconds (except for pool and VFS2), whereas random testing techniques reach a saturation point after 200 seconds in most cases. In addition, when reaching a saturation point, the number of SP coverage requirements covered for TSA is always greater than that of the random testing techniques. Moreover, TSA works for all our subjects, whereas some random techniques, such as RND-s100, do not work for the Java Server programs (i.e, cache4j,pool, and VFS2). This observation is one of the most important observations in the paper because our testing technique can be used practically instead of random testing techniques [5, 18], which are used in industry [1].

3.4 Study 3: Precision of Estimation

The goal of the study is to investigate how precisely our technique estimates a set of SP coverage requirements. To do this, we

- Ran the estimation phase on each subject and, for each subject, created a set of SP coverage requirements
- Using the sets of SP coverage requirements, ran TSA 30 times

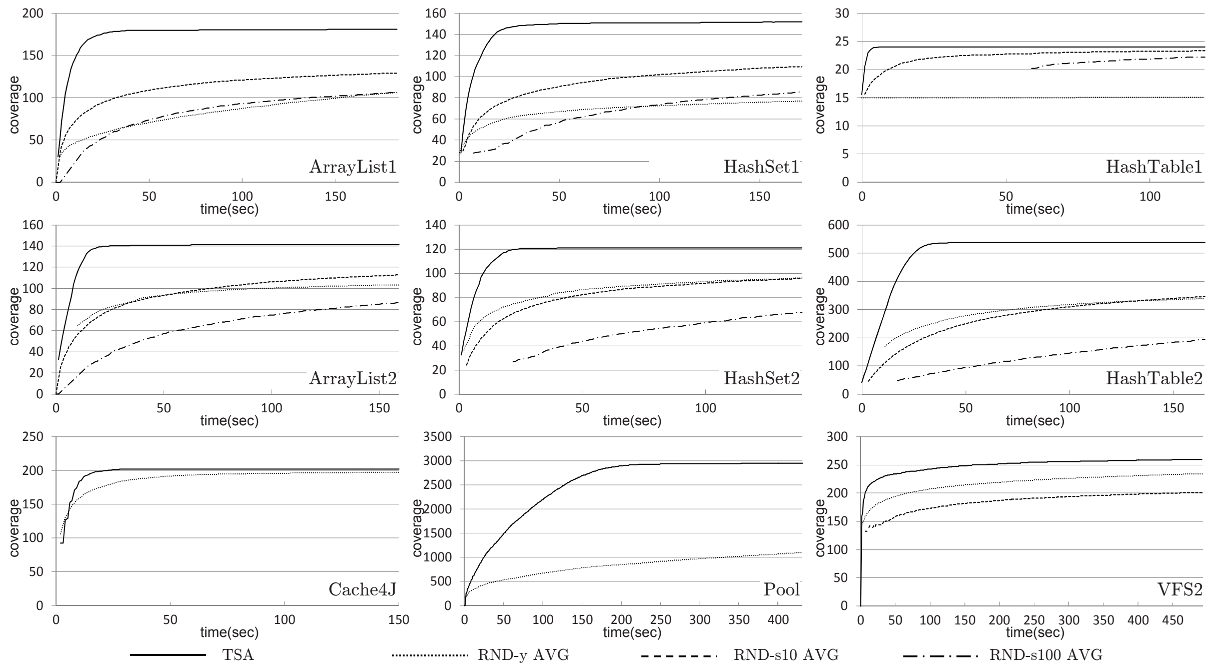

Figure 5: Results of Study 2 (Efficiency): The number of covered SP requirements until the saturation point of `TSA`

Table 3: Results of Study 2 (Efficiency).

Program	RND-y	RND-s10	RND-100	TSA
	Cvg. / Time	Cvg. / Time	Cvg. / Time	Cvg. / Time
ArrayList1	129.5 / 411.5	173.0 / 457.7	177.9 / 666.1	181.2 / 184.2
ArrayList2	106.2 / 278.2	141.2 / 372.8	140.0 / 550.3	141.4 / 159.7
HashSet1	82.3 / 300.8	152.4 / 481.6	150.4 / 629.7	151.7 / 172.4
HashSet2	103.8 / 296.9	119.7 / 336.5	116.9 / 505.1	120.8 / 139.3
HashTable1	17.0 / 145.0	24.0 / 136.8	23.8 / 162.4	24.0 / 120.0
HashTable2	-/-	472.0 / 778.0	488.6 / 1536.6	538.0 / 165.4
LinkedList1	118.6 / 467.9	172.0 / 437.9	177.6 / 727.8	181.2 / 155.0
LinkedList2	108.8 / 289.6	141.1 / 457.7	138.9 / 644.5	141.2 / 161.2
TreeSet1	81.1 / 284.9	151.5 / 427.7	151.1 / 659.4	151.4 / 191.2
TreeSet2	109.6 / 316.7	117.4 / 356.9	113.0 / 545.8	120.5 / 139.8
cache4j	199.0 / 231.0	-	-	202.2 / 146.1
pool	888.0 / 388.5	-	-	2950.5 / 431.4
VFS2	235.3 / 499.2	199.1 / 599.1	-	260.1 / 493.9

Table 4: Results of Study 3 (Precision of Estimation).

Program	Estimated # of requirements	Requirements covered	False positives	False negatives
ArrayList1	181	182	0	1
ArrayList2	145	144	2	1
HashSet1	139	154	0	15
HashSet2	116	121	1	6
HashTable1	23	24	0	1
HashTable2	597	540	57	0
LinkedList1	181	182	0	1
LinkedList2	146	144	2	0
TreeSet1	152	154	0	2
TreeSet2	118	121	1	4
cache4j	213	204	9	0
pool	5292	3416	1886	10
VFS2	335	264	77	6

– For each of the 30 runs, executed the program 500 times and recorded the accumulated covered SP coverage requirements
– Took the union of the accumulated SP coverage requirements over the 30 runs
– Compared the estimated SP requirements to the accumulated SP coverage requirements

Table 4 shows the results of this study. The first column shows the program name. The second column shows the estimated number of SP coverage requirements, computed in the estimation phase. The third column shows the number of covered SP coverage requirements. The fourth and fifth columns show the number of false positives (i.e., estimated but not covered in testing executions) and false negatives (i.e., not estimated but covered in testing executions), respectively.

The result shows that the estimation is precise with less than 2% difference for all subject except `HashTable2`, `pool`, and `VFS2`. To find the cause of the imprecision, we manually inspected the code and the SP requirements to find the causes of false positives and false negatives.

One possible reason for the false positives is that our estimation technique is not precise enough to filter our infeasible coverage requirements. We used the lockset method to remove the likely-infeasible coverage requirements, but we may leverage another method, such as the happens-before relationship, to improve the precision.

There are several reasons for false negatives. Because the estimation technique is a dynamic technique, it explores only a limited space during runtime, and can miss some critical observations that do not appear in the estimation phase. One source of false negatives we found is that synchronization operations, such as locks, inside a loop do not appear in the estimation execution, but do appear in testing execution. Another source of false negatives we found is an aliasing problem. A lock statement appearing in a thread can be paired with an access of the other thread and constitute a coverage requirement, but in other executions, it can constitute another coverage requirement by coupling with another thread.

Table 5: Results of Study 4 (Impact of the Estimation based Heuristic).

Program	TSA-*h*	TSA	% of Rule 3
	Coverage / Time	Coverage / Time	executed
ArrayList1	177.6 / 274.4	181.2 / 184.2	81.9 %
ArrayList2	130.8 / 246.3	141.4 / 159.7	98.0 %
HashSet1	151.3 / 271.5	151.7 / 172.4	81.9 %
HashSet2	98.0 / 198.9	120.8 / 139.3	98.1 %
HashTable1	23.7 / 120.0	24.0 / 120.0	16.9 %
HashTable2	539.6 / 388.8	538.0 / 165.4	95.0 %
LinkedList1	179.9 / 278.2	181.2 / 155.0	86.1 %
LinkedList2	129.9 / 237.7	141.2 / 161.2	98.0 %
TreeSet1	151.6 / 258.4	151.4 / 191.2	82.4 %
TreeSet2	98.8 / 237.5	120.5 / 139.8	97.9 %
cache4j	201.9 / 205.8	202.2 / 146.1	99.9 %
pool	-	2950.5 / 431.4	91.9 %
VFS2	246.7 / 478.2	260.1 / 493.9	83.9 %

3.5 Study 4: Impact of the Estimation Based Heuristic

The goal of the study is to investigate the impact of the estimation based heuristic on the efficiency of the testing phase. To do that, we implemented a tool called `TSA-`*h* that removes the Rule 3 part of Algorithm 1 in `TSA`. To do this study, using the implementation of our technique, we

- Ran the estimation phase on each subject and, for each subject, created a set of SP coverage requirements
- Using the sets of SP coverage requirements, ran `TSA` and `TSA-`*h* 30 times
  - For each of the 30 runs, executed the program for 30 minutes and recorded the accumulated number of covered SP coverage requirements for each execution
  - Computed the saturation point for each run
  - Computed the average saturation point (i.e., a pair of the average time and the average SP coverage requirements of the saturation points) over 30 runs

Table 5 shows the results of the study. The first column shows the program name. The second and third columns show the number of covered SP coverage requirements and the time for `TSA-`*h* and `TSA`, respectively. The fourth column shows the ratio of the application of Rule 3 over Rule 1, Rule 2 and Rule 3 in Algorithm 1 (lines 15–30). For example, for `ArrayList1`, `TSA-`*h* reaches saturation within 274.4 seconds and covers 177.6 SP coverage requirements, whereas `TSA` reaches saturation within 184.2 seconds and covers 181.2 SP coverage requirements using `TSA`. For `TSA`, Rule 3 was applied 81.9% of time when to remove an action p' out of *paused* on average.

The results show that `TSA` covers a greater number of SP coverage requirements than `TSA-`*h* at all times at the saturation point. In addition, `TSA` reaches the saturation point faster than `TSA-`*h*. These two observations imply that the estimation based heuristic contributes to the efficiency of `TSA` substantially for our subjects.

Another important observation is that as the ratio of application of Rule 3 increases, `TSA` performs better than `TSA-`*h*. For example, for `HashTable1`, the ratio of application of Rule 3 is only 16.9%, and the differences of coverage and time of the two techniques are minimal. In contrast, Rule 3 was applied 99.9% of time for `cache4j`, and there is a huge gap of performance between `TSA` and `TSA-`*h*. This observation also supports that the estimation based heuristics is the key asset of our technique.

3.6 Threats to Validity

Threats to external validity limit the extent to which our results will generalize to other concurrent programs. The main external threats include the subject programs we used. The programs may not represent all types of concurrent programs. To mitigate the problem, we included several kinds of programs, from library classes to server applications.

Threats to internal validity include the correctness of our prototype implementation. Our prototype may contain unknown bugs, which can change the results of the empirical studies. To reduce the risk of the bugs, we minimized our implementation by building our tool on top of the publicly available CalFuzzer tool, which was used in many previous studies [6, 7, 16]. Moreover, we iterated the development with rigorous code reviews. Another threat to internal validity is that we used our implementation of random testing techniques instead of existing tools.

4. RELATED WORK

There are two main areas of research related to ours: testing for concurrent programs and coverage criteria for concurrent programs.

The simplest, but most practical, type of technique for testing concurrent programs is random testing. Random testing runs the program many times while injecting artificial delays into thread schedules to produce different interleavings. The delay-injection technique increases the likelihood of revealing concurrency bugs. ConTest [5] is based on random-delay injection with a focus on injecting delays at program points related to buggy code patterns. Rstest [18] performs escape analysis to identify thread-escaping code points, and to inject delays at those points at runtime. These techniques produce independent random interleavings for different runs. Thus, there may be many duplicate interleavings, which may be inadequate for covering previously uncovered interleavings. In contrast, our technique leads the thread interleaving toward covering previously uncovered interleavings as the number of runs increases.

Another type of testing for concurrent programs is based on concurrency bug analysis. These techniques identify potential concurrency bugs using static analysis [7] or using dynamic analysis obtained from a program trace [14]. The techniques then run the program while manipulating the thread scheduler to trigger the possible bugs. CalFuzzer [6] is a Java-based software framework that implements program analysis and testing for concurrent programs. RaceFuzzer [16] and DeadlockFuzzer [7], which were built using CalFuzzer, target revealing data races and deadlocks, respectively. Our technique also uses the CalFuzzer framework. However, unlike these techniques, our technique is not restricted to manipulating interleavings near possible buggy code points, but explores interleavings of any program point.

A third type of testing is systematic testing, which explores distinct interleavings of the program in each different run. The inherent problem of these techniques is that the interleaving space to explore is exponentially large [12]. CHESS [12] uses preemption-bounded model checking, which explores interleaving with few preemptions (e.g., preemptions less than 2). Wang et al. [20] propose an interleaving exploration method, which is likely to cover previously uncovered statement-pair coverage. Techniques in this category are guaranteed not to investigate the same interleaving for different runs, but the techniques incurs significant overhead. For example, the most recent technique [20] incurs $10\times$ to $100\times$ execution overhead. In contrast, our technique incurs low overhead and is always faster than random testing techniques [5, 18] (see Section 3.3).

Regarding coverage criteria, two essential questions are (1) how to achieve higher coverage faster, and (2) how much testing is enough to guarantee quality. For sequential programs, both questions are well researched. For example, regarding Question 1, testing techniques using symbolic execution [8, 21] explore a program to increase coverage faster. Regarding Question 2, there are some industrial standard metrics to guarantee quality of the software. However, neither question is well-researched for concurrent programs.

For Question 1, there are some coverage criteria suggested for concurrent programs [3, 11, 19, 22]. The coverage criteria capture the behavior among threads. Examples of the criteria include def-use pair coverage [22], synchronization-pair coverage [19], and event-pair coverage criteria [3]. Lu et al. [11] discuss several coverage criteria and determine a subsumption hierarchy among them. Letko et al. [10] performed empirical studies with the above criteria, and show that some criteria are better for different testing techniques. However, there is little research aimed at achieving higher coverage faster based on those criteria to address Question 1 directly. ConTest [5, 19] utilizes the coverage feedback for determining the amount of noise to inject for bug detection. In contrast, our technique is one of the first techniques that directly control thread scheduling to increase coverage, and specifically aims at high synchronization-pair coverage faster. Although, we applied our technique for only one type of coverage criteria, it can be easily extended to other criteria, and we plan to do more studies on them in the future.

Regarding Question 2, there is little research to determine the thoroughness of testing for concurrent programs. One representative technique is saturation-based testing [17] that monitors the number of executed coverage requirements until the rate of increase of covering new requirements is less than a threshold (i.e., the coverage reaches a saturation point). This testing adequacy criteria is often used in testing for concurrent programs [10], and we use it as the stopping criterion in our empirical studies in Section 3.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented a new thread-schedule technique that aims at achieving high test coverage for concurrent programs. Our empirical studies show that our technique is more effective and efficient in achieving high synchronization coverage than existing random testing techniques for a suite of Java programs.

Although our initial studies are promising, there are several areas of future work that we will pursue. First, although there is a body of research on investigating the relationship between coverage and fault-detection ability for sequential programs [4, 13, 15], there is little work that evaluates this issue for concurrent programs. Thus, we will perform additional empirical studies on this relationship. Second, we will perform additional studies on more and varied programs to determine whether our technique generalizes. Finally, our studies used SP coverage as the criterion because it is one of the basic coverage criteria for concurrent programs [11]. However, there are other criteria for which our technique may perform well, and we plan to extend it to these other criteria.

6. ACKNOWLEDGEMENTS

This work was supported by the ERC of MEST/NRF of Korea (Grant 2012-0000473) and NSF CCF-1116210.

7. REFERENCES

[1] ConTest: A tool for testing multi-threaded java applications. `https://www.research.ibm.com/haifa/projects/verification/contest/`.

[2] Our tool, subjects, and the results of the studies. `http://pswlab.kaist.ac.kr/data/issta12.zip`.

[3] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Symp. Princ. Prac. of Paral. Prog. (PPoPP)*, 2005.

[4] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Int'l Works. Adv. Model-based Test. (A-MOST)*, pages 1–7, 2005.

[5] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. In *Conf. Java Grande*, 2001.

[6] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: an extensible active testing framework for concurrent programs. In *Int'l Conf. Comp. Aid. Veri. (CAV)*, 2009.

[7] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, (6), 2009.

[8] M. Kim, Y. Kim, and G. Rothermel. Scalable distributed concolic testing approach: An empirical evaluation. In *Int'l Conf. Softw. Test. Verif. Valid. (ICST)*, 2012.

[9] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for Search-Based testing of concurrent software. In *Works. Paral. Dist. Sys. Test. Analy. Debug. (PADTAD)*, 2010.

[10] Z. Letko, T. Vojnar, and B. Křena. Coverage metrics for saturation-based and search-based testing of concurrent software. In *Int'l Conf. Run. Veri. (RV)*, 2011.

[11] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, pages 533–536, 2007.

[12] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, (6), June 2007.

[13] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Int'l Symp. Softw. Test. Analy. (ISSTA)*, pages 57–68, 2009.

[14] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Int'l Conf. Arch. Supp. Prog. Lang. Oper. Sys. (ASPLOS)*, 2009.

[15] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Int'l Conf. Softw. Eng. (ICSE)*, pages 287–301, 1993.

[16] K. Sen. Race directed random testing of concurrent programs. In *Int'l Conf. Prog. Lang. Desig. Impl. (PLDI)*, 2008.

[17] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2009.

[18] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Runt. Veri. Works. (RV)*, 2002.

[19] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *Par. Dist. Sys. Test. Analy. Debug. (PADTAD)*, 2009.

[20] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Int'l Conf. Softw. Eng. (ICSE)*, 2011.

[21] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Symp. Found. of Softw. Eng (FSE)*, 2010.

[22] C. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *Int'l Symp. Softw. Test. Analy. (ISSTA)*, pages 153–162, 1998.