Performance Regression Testing of Concurrent Classes

Michael Pradel
EECS Department
University of California, Berkeley

Markus Huggler
Dept. of Computer Science
ETH Zurich

Thomas R. Gross
Dept. of Computer Science
ETH Zurich

ABSTRACT

Developers of thread-safe classes struggle with two opposing goals. The class must be correct, which requires synchronizing concurrent accesses, and the class should provide reasonable performance, which is difficult to realize in the presence of unnecessary synchronization. Validating the performance of a thread-safe class is challenging because it requires diverse workloads that use the class, because existing performance analysis techniques focus on individual bottleneck methods, and because reliably measuring the performance of concurrent executions is difficult. This paper presents SpeedGun, an automatic performance regression testing technique for thread-safe classes. The key idea is to generate multi-threaded performance tests and to compare two versions of a class with each other. The analysis notifies developers when changing a thread-safe class significantly influences the performance of clients of this class. An evaluation with 113 pairs of classes from popular Java projects shows that the analysis effectively identifies 13 performance differences, including performance regressions that the respective developers were not aware of.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Performance, Measurement, Algorithms

Keywords

Thread safety, Test generation, Performance measurement

1. INTRODUCTION

A thread-safe class is a class that encapsulates all synchronization necessary to allow multiple concurrent threads to access a shared object without further synchronization. Clients of the class can access an instance from multiple threads as if there was no other thread using the same instance. Because correctness must be guaranteed by the class developer, thread-safe classes simplify the task of parallel programming for the clients. In many multi-threaded programs, a small number of thread-safe classes implement most concurrency-related functionality. Since these classes are at the core of the program, their behavior is crucial for the entire program.

Developers of a thread-safe class aim for two contradicting goals. On the one hand, the class must be correct, in particular, it should indeed be thread-safe. To ensure thread safety, the class must synchronize all concurrent accesses to an instance of the class, for example, with synchronized methods. On the other hand, the class should perform well, even when many threads access a shared instance concurrently. To ensure good performance, thread-safe classes typically try to minimize the frequency and granularity of synchronization, for example, by using short synchronized blocks instead of synchronized methods. Due to the contradicting nature of these two goals, correctness and performance, developers often struggle to achieve both.

To achieve correctness, developers can rely on bug finding techniques [17, 16, 6, 46, 26, 41] and approaches that influence the schedule of concurrent executions [15, 36, 12, 7, 45, 40, 32, 30, 25, 57, 49]. In contrast, developers currently have only little support to measure, improve, and maintain the performance of thread-safe classes. Traditional CPU profiling [21] is only of limited help, as it focuses on individual bottleneck methods. Furthermore, performance issues are relatively hard to identify because measuring the performance of a concurrent execution in a reliable way is non-trivial. Recent work on automatically finding performance problems focuses on particular bug patterns [54, 55, 38] but does not address concurrency-related performance problems.

This paper presents SpeedGun, an automatic performance regression testing technique for thread-safe classes. The approach allows developers to easily assess how a change in a thread-safe class influences the performance of its clients. The key idea is to automatically generate concurrent performance tests that exercise the class in various ways and to measure the performance before and after a change. SpeedGun warns developers if a change significantly decreases the performance, that is, if the perceived slowdown is above a configurable threshold. SpeedGun also reports performance improvements to enable developers to verify whether a sup-

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07 ...$15.00.

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized) {
      this.initialized = true;
    }
  }
  boolean isInitialized() {
    return this.initialized;
  }
}
```

(a) Before October 2007: Class is not thread-safe because reads and writes of `initialized` are not synchronized.

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!isInitialized()) {
      setInitialized(true);
    }
  }
  synchronized boolean isInitialized() {
    return this.initialized;
  }
  synchronized void setInitialized
      (boolean b) {
    this.initialized = b;
  }
}
```

(b) October 2007: Fix thread safety problem by making methods synchronized. Leads to performance regression reported in May 2009.

```
class ExpandoMetaClass {
  private volatile boolean initialized;
  synchronized void initialize() {
    if (!isInitialized()) {
      setInitialized(true);
    }
  }
  boolean isInitialized() {
    return this.initialized;
  }
  void setInitialized(boolean b) {
    this.initialized = b;
  }
}
```

(c) September 2009: Fix performance regression by replacing synchronized methods with volatile variables.

Figure 1: Three versions of a thread-safe class from Groovy.

posed optimization works as expected.

Our approach is enabled by two components. First, we present a generator of concurrent performance tests, that is, tests that run for a significant amount of time and have a high degree of concurrency. The generated tests focus on those parts of a class under test (CUT) that have been changed from one version to another. Generating such tests goes beyond the capabilities of existing generators of concurrent tests [41, 37], which focus on short tests that expose correctness bugs. Furthermore, our approach addresses the problem of automatically finding a reasonable length of tests for a particular CUT. Performance tests must be long enough to allow for reliably measuring performance but should not unnecessarily protract the testing process.

Second, we present a component that, given a set of concurrent performance tests, compares the performance of two versions of the CUT and decides whether to report a performance difference. For each generated test, this component directly compares the two versions by measuring their performance for the same test. This part of the approach incorporates best practices for reliably measuring the performance of concurrent Java programs.

As a motivating example, consider the class `ExpandoMetaClass` from the Groovy project (Figure 1). In a bug report from October 2007, a user complained about a thread safety problem caused by missing synchronization.[1] The developers fixed the problem by making two methods of the class synchronized. Unfortunately, this change significantly decreased the performance of the class, and in May 2009, a user complained that "as the number of concurrent requests increases response times increase dramatically".[2] In response to the second bug report, the developers replaced the course-grained synchronization introduced in 2007 by more fine-grained synchronization via a volatile field. 19 months of suboptimal performance passed between the first change that degraded performance and the second change that restored it. As the example illustrates, developers of thread-safe classes are often unaware of performance changes that come as a side effect of fixing correctness problems.

SpeedGun supports developers by finding performance regressions and by confirming supposed performance improvements. For the above example, SpeedGun could have helped to detect the performance regression right after introducing it. For the first change, which introduces synchronized methods, the analysis reports a performance regression. In contrast, adding the more fine-grained synchronization of Figure 1c would not have triggered a regression report because the performance is similar. Besides diagnosing regressions, SpeedGun helps to verify supposed performance improvements. For the second change, which replaces coarse-grained synchronization with fine-grained synchronization, the analysis reports a performance improvement. Of course, performance problems are not always as easy to spot as in Figure 1. In the evaluation of this work, we encounter more intricate examples, where a quick manual analysis cannot accurately assess the performance impact of a change.

SpeedGun is designed to provide an automatic tool that is easy to use in practice. To this end, the approach has three properties. First, SpeedGun requires no input except for two versions of the CUT, possibly accompanied by helper classes that the CUT depends on. Second, the output of SpeedGun is precise in the sense that each reported performance difference has been observed in executions of the CUT and has been found to be larger than a configurable threshold. Each report comes with a generated test suite to reproduce the performance difference. Third, SpeedGun relieves developers from dealing with the difficulties of measuring the performance of concurrent executions. Such performance measurements are challenging, for example, because different thread interleavings may expose different performance properties and because just-in-time compilation and memory management influence performance in surprising ways. Our approach incorporates various best practices for measuring performance and lets developers benefit from them without any effort.

To evaluate our approach, we apply SpeedGun to 113 pairs of thread-safe classes from four popular Java code bases, where each pair consists of a class before and after a change committed by the respective developers. Among the 113 pairs of classes, the approach effectively finds 13 performance differences, including the differences illustrated in Figure 1. The reported performance differences mostly coincide with a baseline established by manually inspecting the

[1] Issue 2166 in Groovy's bug database.
[2] Issue 3557 in Groovy's bug database.

pairs of classes. Moreover, the analysis reveals performance changes that the developers of the classes were apparently not aware of when changing the class, and it casts doubts about some performance improvements expected by the developers.

In summary, this paper contributes the following:

- A generator of concurrent performance tests. It efficiently creates long-running tests that have a high degree of concurrency and that allow for comparing the performance of two versions of a class.
- Algorithms for repeatedly executing generated tests and for reliably measuring their performance to decide whether a change influences the performance of a CUT in a significant way. This part of the approach could also be used independently of the first contribution, for example, with manually written performance tests.
- An implementation to yield a practical tool and evidence of its effectiveness. We show that the approach is useful to detect performance regression bugs and to verify supposed performance improvements in real-world classes.

2. CHALLENGES FOR ACCURATE PERFORMANCE MEASUREMENT

A major challenge for automatic performance regression testing is how to accurately measure performance. Since addressing this challenge influences the design of the entire approach, we discuss in the following the various sources of non-deterministic performance of concurrent Java programs.

- *Measurement accuracy.* The JVM provides a nanosecond timestamp, but its accuracy varies among platforms. Our approach assumes a *minimal measurable time span* t_{min} that can be measured accurately despite inaccurate timestamps, and it creates tests that run longer than t_{min}.
- *Thread scheduling.* Different executions of a concurrent test may lead to different thread interleavings, which in turn may expose different performance properties. We mitigate the effect of different thread schedules by repeatedly executing each generated test at least r_{min} times, allowing the scheduler to trigger different thread interleavings and performance properties. Existing techniques for forcing different schedules [15, 7, 36, 12] perturb the performance of the scheduled program and therefore are not applicable in our approach.
- *Just-in-time (JIT) compilation.* The JIT compiler may optimize tests at arbitrary points while they are running and change their performance. To deal with the influence of JIT compilation, we repeatedly run tests during a *warm-up phase* and measure performance afterwards, during a *steady-state phase*. This approach is common practice for measuring performance in a managed runtime [19].
- *Garbage collection.* Memory management can influence the performance of a Java program [19]. To deal with this influence, we trigger the garbage collector before starting a measurement. Although there is no standardized way to force JVMs into running the garbage collector, this best effort approach works well in practice.

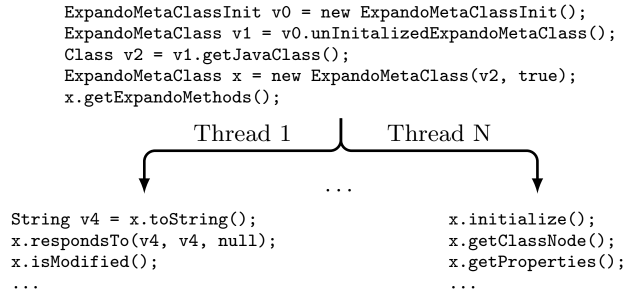

Figure 2: Example of a generated test. The prefix (above) creates an instance `x` of the CUT. The suffixes (below) use `x` concurrently.

3. GENERATING CONCURRENT PERFORMANCE TESTS

The first part of our performance regression testing technique is a generator of concurrent performance tests. The generator takes two versions of a class as its input and creates tests that exercise the common interface of both classes. SpeedGun creates *concurrent tests* that consist of a sequential *prefix* followed by multiple concurrently executing *suffixes*. Each prefix and suffix is a sequence of method calls and field accesses, whose result (if any) is stored in a fresh local variable. The prefix instantiates the CUT and calls methods on the created instance to bring it into a state that may expose performance problems. Suffixes run after the prefix, each in a separate thread, and are executed concurrently with each other. The instance of the CUT created in the prefix is shared by all suffixes.

Figure 2 shows a generated performance test for `ExpandoMetaClass`. The upper part is the prefix that creates an instance `x` of the CUT. Before creating the shared instance `x`, the prefix contains several other method calls that provide arguments for creating `x`. The lower part shows parts of the concurrent suffixes that exercise the shared instance `x`.

To accurately measure the performance of concurrent tests, it is crucial to have tests that run for a significant amount of time. In particular, the concurrent part of a test must be significantly longer than the minimal measurable time span t_{min} (Section 2). During our evaluation, tests typically have suffixes with several thousands of method calls to meet this requirement. Another requirement is that performance tests should have many concurrent threads to simulate the degree of contention that a thread-safe class may experience in real-world software. The need for long-running tests with many threads distinguishes SpeedGun from existing approaches for generating concurrent tests [41, 37], which create short tests with a small number of threads to find concurrency-related correctness problems. Naively extending these existing test generation approaches to create long tests does not scale. Furthermore, repeatedly executing short tests does not work well because the overhead for creating and joining threads often outweighs differences in

the concurrent performance of two versions of the CUT.

Our approach to create concurrent performance tests has three steps. First, SpeedGun analyzes the two versions of the CUT and selects a set of methods to focus on. Second, SpeedGun determines how long the tests for the given CUT must be to allow for accurately measuring performance. This step is important to obtain accurate measurements without unnecessarily protracting the testing process by creating overly long tests. Third, SpeedGun uses feedback-directed random test generation to create a set of concurrent tests. The remainder of this section elaborates on these three steps.

3.1 Choosing Methods Under Test

To compare the performance of two versions of a CUT, SpeedGun generates tests that run with both versions. For this purpose, the generator considers a method only if the method is provided in the public interface of both classes. As a result, the generated tests can test either version of the class, depending on which class is loaded. We further extend the test generator so that it focuses on methods and fields that changed from the old to the new version of the CUT. We call all methods that are directly or indirectly influenced by these changes *focus methods* and prioritize them during test generation. With this prioritization, the generated tests exercise focus methods more intensively than other methods. This setup allows our approach to find performance problems introduced in the new version of the class.

Instead of prioritizing focus methods, the test generator could ignore methods that are not influenced by the changes. However, because some methods of a CUT may require other methods to be called first (for example, to establish a precondition), the test generator also calls non-focus methods.

3.2 Determining the Length of Tests

The length of suffixes that the test generator creates is a crucial parameter of our approach. Too short suffixes lead to inaccurate performance measurements and the approach cannot correctly identify performance differences. Too long suffixes will waste time while creating and executing tests. Instead, SpeedGun should repeat tests as often as possible within a given time budget because each repetition can potentially trigger a different interleaving, that is, expose different performance properties. Unfortunately, the relation between the running time of a test and the length of its suffixes is CUT-specific. For example, a test of a particular length may take longer for a CUT where each method is synchronized than for a CUT that uses fine-grained locking.

To relieve developers from manually setting the length of tests, SpeedGun determines a reasonable length for a particular CUT with an automatic heuristic analysis. The analysis assumes a fixed length t_s of the steady-state phase during which SpeedGun measures performance and a minimal number r_{min} of times that a test should be repeated to expose diverse performance properties. Let r_T be the number of times one can repeat a test T within t_s, and let $\bar{t}_T$ be the average time taken to execute T. Then the following three (in)equalities describe the constraints within which to search for a reasonable test length:

$$t_s \approx r_T \cdot \bar{t}_T \quad (1)$$

$$r_T > r_{min} \quad (2)$$

$$\bar{t}_T > t_{min} \quad (3)$$

Algorithm 1 Find a reasonable length of tests for a CUT.

Input: Old and new version of the CUT
Output: Length of suffixes for testing the CUT

1: $l_{min} \leftarrow startMin$
2: $l_{max} \leftarrow startMax$
3: **while** $l_{min} < l_{max}$ **and** $tries < maxTries$ **do**
4: $\quad l_{current} \leftarrow (l_{min} + l_{max})/2$
5: $\quad T \leftarrow$random test with suffixes of length $l_{current}$
6: $\quad \bar{t}_{T1}, r_{T1} \leftarrow$ execute T with old version
7: $\quad \bar{t}_{T2}, r_{T2} \leftarrow$ execute T with new version
8: $\quad tooShort \leftarrow \bar{t}_{T1} \leq t_{min}$ **or** $\bar{t}_{T2} \leq t_{min}$
9: $\quad tooLong \leftarrow r_{T1} \leq r_{min}$ **or** $r_{T2} \leq r_{min}$
10: $\quad$**if** $tooShort$ **and** $tooLong$ **then**
11: $\quad\quad$Abort (cannot find a reasonable length)
12: $\quad$**else if** !$tooShort$ **and** !$tooLong$ **then**
13: $\quad\quad$**return** $l_{current}$ ▷ Found a reasonable length
14: $\quad$**else if** $tooShort$ **then**
15: $\quad\quad l_{min} \leftarrow l_{current}$ ▷ Try longer tests
16: $\quad$**else if** $tooLong$ **then**
17: $\quad\quad l_{max} \leftarrow l_{current}$ ▷ Try shorter tests
18: $\quad$**end if**
19: **end while**
20: Abort (cannot find a reasonable length)

(1) ensures to repeat the test r_T times within the steady state phase. (2) ensures that the test gets at least the minimal number of repetitions. (3) ensures that executing a test takes long enough to measure its performance, on average.

Given these generic constraints and a particular CUT, SpeedGun performs a binary search for a reasonable test length (Algorithm 1). The algorithm repeatedly creates a random test (line 5, details in Section 3.3) and executes it for both versions of the CUT (lines 6 and 7). We measure how long the test takes on average for each version, $\bar{t}_{T1}$ and $\bar{t}_{T2}$, and how many repetitions are possible within t_s, r_{T1} and r_{T2}. For these measurements, we repeatedly execute the test until the total time exceeds t_s. If the average test running time is below the minimal measurable time t_{min} for one of the two versions, then the algorithm explores larger test lengths (line 16). Likewise, if the number of repetitions is below the minimal acceptable number r_{min} of repetitions, the algorithm explores smaller test lengths (line 17). The binary search continues until the algorithm finds a test length that satisfies constraints (1) to (3). To bound the time spent in the algorithm, it is aborted after trying $maxTries$ lengths without finding a length that satisfies the constraints. Similarly, the algorithm aborts the search if it finds a test length to be too small and too large at the same time. If the algorithm fails to propose a reasonable test length, SpeedGun may fall back on a specified maximum test length. During our experiments, the algorithm always succeeds to find a reasonable test length.

The presented approach to find a reasonable test length for a particular CUT is a heuristic and may not work well for all CUTs. A limitation of the approach is that it extrapolates the performance of the CUT from a small number of tests and measurements. As we show in our evaluation, the approach works well for classes from popular libraries.

3.3 Generating Performance Tests

We build upon an existing generator of concurrent unit tests [41].[3] It uses feedback-directed, random test generation, which was first described in [39] for sequential tests. The basic idea is to construct sequences of statements by randomly choosing fields, methods, and method arguments, by executing partial tests, and by extending sequences only if they do not lead to an exception.

SpeedGun requires tests with a large number of suffixes that each contain a large number of method calls. Naively applying feedback-directed random test generation to create such tests is impractical. The reason is that to create a call sequence of length l, the test generator executes partial call sequences of length 1, 2, ..., and l. That is, creating a sequence with l calls requires executing at least $\frac{l^2+l}{2}$ calls. For example, naively generating a suffix with 2,000 calls requires over 2 million calls during the test generation process. Since SpeedGun generates many suffixes that each have thousands of calls, this naive approach does not scale very well.

We address this challenge by creating smaller suffixes and by repeating them to obtain tests of the desired length. To obtain a suffix of length l, the approach creates a call sequence of length $\sqrt{l}$ and repeats it $\sqrt{l}$ times. This approach is a pragmatic compromise between creating diverse tests and taking a reasonable amount of time for generating tests.

To create tests with a high degree of contention, SpeedGun supports an arbitrary number of threads and creates tests with a configurable number N of suffixes. We expect users of the approach to choose N depending on typical usage scenarios of the CUT.

4. TEST EXECUTION AND PERFORMANCE MEASUREMENT

Given a generated performance test, the analysis executes the same test for both versions of the CUT and measures the performance. As described in Section 2, measuring the performance of a concurrent Java program is challenging. We address this challenge by repeatedly measuring the steady-state performance of each test and by only accepting measurements that vary within specified bounds. In summary, the test execution and performance measurement component of the approach consists of four steps:

1. Estimate how often to repeat the test to fill a specified warm-up period t_w and a specified steady-state period t_s.
2. Randomly decide which of the two versions to test first.
3. Gather execution times for one version.
4. Gather execution times for the other version.

To run a test for specified warm-up and steady-state periods, SpeedGun repeatedly executes it. Repeating a test more often for one version than for the other version can influence the measured performance because the JIT compiler may optimize a method only after it has been called a particular number of times. To avoid skewing the measured performance, our first step is to estimate a single number of repetitions to be used for both versions. To this end, we run the test for both versions for a period t_s and count the repetitions achieved. From these numbers, we compute how often to repeat the test during the warm-up phase and during the steady-state phase, respectively, giving two numbers r_w and r_s that are used for both versions of the class.

[3] `http://thread-safe.org`

Algorithm 2 Gather execution times of a test.

Input: Test T; Number of repetitions r_w and r_s for the warm-up phase and the steady-state phase, respectively
Output: Set $\mathcal{M}$ of execution times or *inconclusive*

```
 1: runGarbageCollection()
 2: repeat(T, r_w)                          ▷ Warm-up phase
 3: M ← ∅                                   ▷ Start of steady-state phase
 4: repeat
 5:     M ← M ∪ repeatAndMeasure(T, r_s)
 6: until m_min measurements done
 7: while σ(M) > mean(M) · σ_stop do
 8:     M ← M ∪ repeatAndMeasure(T, r_s)
 9:     if |M| = m_max then
10:         if σ(M) ≤ mean(M) · σ_acceptable then
11:             return M
12:         else
13:             return inconclusive
14:         end if
15:     end if
16: end while
17: return M                                ▷ End of steady-state phase
```

The second step is to randomly decide which of the two versions to test first. A random decision avoids biasing the measurements towards one of the two versions. For example, always running the old version before the new version may give a performance advantage to the new version because the executions of the old version may trigger some JIT optimization from which the new version benefits.

The third and the fourth steps, gathering execution times for both versions, are the core of the test execution and performance measurement component. Algorithm 2 summarizes how to gather execution times for a given test T and for given numbers of repetitions r_w and r_s. Before starting any measurements, the algorithm triggers the garbage collector. Then, the algorithm repeats the test r_w times as the warm-up phase. Finally, during the steady-state phase (lines 3 to 17), the algorithm builds a set $\mathcal{M}$ of execution times of the test. The basic idea is to repeatedly measure the time required for executing the test r_s times until the standard deviation $\sigma(\mathcal{M})$ of the measured execution times is below a specified percentage of the mean of the measured execution times, that is, below $\overline{\mathcal{M}} \cdot \sigma_{stop}$. This approach mitigates the various sources of non-deterministic performance by taking additional measurements as long as the measured execution times are spread out over a large range of values. The algorithm starts with a minimum number m_{min} of measurements (lines 4 to 6) and adds measurements until the standard deviation is below the threshold (lines 7 to 16), or until a maximum number m_{max} of measurements is reached. If m_{max} is reached, then the series of measurements is rejected as *inconclusive*, unless the standard deviation is below a threshold $\overline{\mathcal{M}} \cdot \sigma_{acceptable}$.

We have two thresholds, $\overline{\mathcal{M}} \cdot \sigma_{stop}$ to stop measuring immediately and $\overline{\mathcal{M}} \cdot \sigma_{acceptable}$ to accept a series of m_{max} measurements, is to provide a knob to account for unavoidable variations of the measured performance. For example, such variations may occur if the machine used for performance

Algorithm 3 *repeatAndMeasure*(T, r)

Input: Test T; Number of repetitions r
Output: Execution time t

```
 1: t ← 0
 2: repeat
 3:    Execute prefix of T
 4:    Setup threads for suffixes of T
 5:    start ← currentTime()              ▷ Start measurement
 6:    for each thread do
 7:       Execute a suffix of T
 8:    end for
 9:    t ← t + currentTime() − start      ▷ Stop measurement
10:    Clean up threads
11: until r repetitions done
12: return t
```

measurements runs other CPU-consuming processes.

4.1 Measurement Scopes

Algorithm 2 uses a function *repeatAndMeasure*() to repeatedly execute a test and to measure its performance. Algorithm 3 shows our default implementation of *repeatAndMeasure*(). The algorithm repeatedly executes the prefix and the suffixes of the test and measures the time from starting the suffixes until all suffixes have terminated. That is, the measurement excludes the time to set up threads and the time for executing the prefix. The rationale to exclude these computations is that they are independent of the CUT's concurrent performance. Alternatively to the measurement scope in Algorithm 3, one can measure the time spent for executing each individual suffix by moving the measurement statements into the for-loop, that is, around line 7. The main difference between these two measurement scopes is that Algorithm 3 includes the time until the last suffix has terminated, whereas the alternative measurement scope considers each suffix in isolation. Furthermore, the alternative measurement scope has a higher measurement overhead because it involves more measurements.

Depending on how the CUT is used in a program, both measurement scopes may represent the CUT's perceived concurrent performance. If the program splits a sequential task into parallel subtasks that all use the CUT, the program may have to wait for all subtasks to finish before making progress. In this case, the approach described in Algorithm 3 reflects what developers are interested in. In contrast, if the program has multiple, independent threads that use the CUT, then there is no need to wait until all suffixes finish and the alternative measurement scope is more appropriate. Our implementation supports both approaches. We focus on Algorithm 3 in the remainder of the paper.

4.2 Inconclusive Tests

Some performance tests may lead to inconclusive results. SpeedGun ignores these measurements when deciding whether to report a performance difference. There are three reasons for inconclusive tests. First, a test execution may be inconclusive because the test raises an exception or leads to a deadlock. Such misbehavior may, for example, be due to a thread safety problem in the CUT, such as the bug in Figure 1a. Second, we reject a test execution as inconclusive if the number of repetitions r_s, which is necessary to fill the steady-state period, is below r_{min}. The rationale for rejecting such tests is that a small number of repetitions may result in a non-representative sample of the performance space of the test. Third, Algorithm 2 may reject a series of measurements as inconclusive if the standard deviation is above the threshold after reaching the maximum number of measurements.

4.3 Test Oracle

The final step of SpeedGun is a test oracle that analyzes the execution times measured for all conclusive tests to decide whether to report a performance difference between the two version of the CUT. For each test and version of the CUT, the test execution and performance measurement component produces one set of execution times. That is, for each test T_i in a set of tests $\mathcal{T} = \{T_1, \cdots, T_n\}$, there are two sets $\mathcal{M}_{i,old}$ and $\mathcal{M}_{i,new}$ of execution times.

For each such test, the oracle decides whether one of the two versions outperforms the other by checking two conditions. First, the oracle checks whether there is a statistically significant difference between the measured execution times. We use the methodology described in [19]: Compute the mean execution time and its confidence interval (confidence level: 98%) for both $\mathcal{M}_{i,old}$ and $\mathcal{M}_{i,new}$, and check whether the confidence intervals overlap. If and only if the confidence intervals do not overlap, the oracle concludes that one version outperforms the other for the given test. Second, the oracle checks whether the measured performance difference is above a threshold δ_{report}. The threshold avoids reporting small differences that developers may not consider relevant, and we discuss its value in Section 5.4. Formally, the oracle considers the old version to outperform the new version if

$$\overline{\mathcal{M}_{i,old}} + \frac{c_{i,old}}{2} < \overline{\mathcal{M}_{i,new}} - \frac{c_{i,new}}{2} \text{ and } \frac{\overline{\mathcal{M}_{i,new}}}{\overline{\mathcal{M}_{i,old}}} - 1 > \delta_{report} \quad (4)$$

and likewise, it considers the new version to outperform the old version if

$$\overline{\mathcal{M}_{i,new}} + \frac{c_{i,new}}{2} < \overline{\mathcal{M}_{i,old}} - \frac{c_{i,old}}{2} \text{ and } \frac{\overline{\mathcal{M}_{i,old}}}{\overline{\mathcal{M}_{i,new}}} - 1 > \delta_{report} \quad (5)$$

where $c_{i,new}$ and $c_{i,old}$ are the sizes of the confidence intervals for $\mathcal{M}_{i,new}$ and $\mathcal{M}_{i,old}$, respectively.

Checking whether one version outperforms the other version partitions the set $\mathcal{T}$ of tests into three subsets: $\mathcal{T}_{none}$, where none of the two versions outperforms the other; $\mathcal{T}_{old}$, where the old version outperforms the new version; and $\mathcal{T}_{new}$, where the new version outperforms the old version. The oracle reports a performance regression if:

$$|\mathcal{T}_{old}| \geq |\mathcal{T}_{none}| \quad \text{and} \quad |\mathcal{T}_{old}| > |\mathcal{T}_{new}| \quad (6)$$

In contrast, the oracle reports a performance improvement if:

$$|\mathcal{T}_{new}| \geq |\mathcal{T}_{none}| \quad \text{and} \quad |\mathcal{T}_{new}| > |\mathcal{T}_{old}| \quad (7)$$

That is, the test oracle informs the developer if one of the two versions outperforms the other in more tests than vice versa, and if this one version outperforms the other in at least as many tests as the number of tests without a clear winner. The rationale for the second condition is to avoid false warnings in cases where many tests are in $\mathcal{T}_{none}$.

Table 1: Classes used in the evaluation and how many of them we manually classify as performance-influencing. For classes marked with *, we analyze the entire version history.

Code base	Class	Methods	Pairs of classes	
			Total	Perf.-infl.
Pool	GenericObjectPool	54–56	3	3
Collections	StaticBucketMap	19	1	1
Groovy	ExpandoMetaClass*	31–71	63	10
JodaTime	DateTime*	35–105	36	1
JodaTime	ISOChronology*	10–12	10	0
Total			113	15

5. EVALUATION

To evaluate the effectiveness of SpeedGun, we apply the implementation to thread-safe classes from real-world Java code bases. We run the analysis on 113 pairs of versions of classes and compare the performance differences reported by SpeedGun to performance differences expected after manual inspection (Section 5.3). Furthermore, we evaluate the sensitivity of SpeedGun to the threshold for reporting performance differences (Section 5.4), and we evaluate the performance of SpeedGun itself (Section 5.5).

5.1 Implementation

Our implementation takes two Jar files, each containing a version of the CUT and helper classes, as its input. Tests are executed reflectively [39, 41], which our initial experiments show to be comparable to normal execution when measuring the relative performance of two versions of a CUT. To run a single test with both versions of the CUT, we modify the class loader so that it loads from the respective Jar file. We use reflective execution to avoid the significant overhead of creating .class files for each generated test and for invoking the JVM for each test execution.

5.2 Experimental Setup

We apply SpeedGun to five thread-safe classes from four Java code bases: Apache Commons Pool, Apache Commons Collections, Groovy, and JodaTime (Table 1). To assess how many relevant warnings the analysis reports, we apply it to pairs of versions of classes with an expected performance difference. Furthermore, to assess whether the analysis reports irrelevant warnings, we apply the analysis to pairs of versions of classes without an expected performance difference. All classes in Table 1 come with manually written test. However, even though all classes are supposed to be used concurrently, only one of them (GenericObjectPool) has multi-threaded tests. All existing tests focus on correctness and do not attempt to measure performance. Since no existing tests or tool can automatically identify performance-related changes in the version histories of these classes, we perform a systematic, manual analysis of the version histories to establish a baseline for evaluating our approach. For the baseline, we expect a performance difference if:

- The developers explicitly mention a change of concurrent performance in the commit message.
- The change explicitly addresses a reported, concurrency-related performance bug.
- The change adds or removes synchronization (for example, by adding or removing the `synchronized` keyword), or the change increases or decreases the scope of synchronization (for example, by replacing a synchronized method with a synchronized block smaller than the method).

To gather pairs of classes for the evaluation, we systematically analyze the version histories of thread-safe classes using two strategies. First, we perform a keyword-based search for bug reports related to concurrency and performance, similar to [28]. For each bug that contains a fix, we include the pair of classes with the buggy class and the fixed class. This first strategy yields pairs of versions for which we expect a performance difference.

Second, we consider the entire version history of three classes and analyze every change of these classes (that is, source code differences, commit messages, and associated bug reports) and classify the change as performance-influencing or not according to the criteria given above. This second strategy yields pairs of versions for which we expect a performance difference and pairs of versions for which we do not expect a performance difference. The inspection of version histories has been done by two of the authors to reduce the risk of misclassifying a change.

Table 1 summarizes the pairs of classes used in the evaluation. For classes marked with *, we consider the entire version history. In total, the marked classes have 154 pairs of versions. We ignore 26 because either the old or the new version fails to compile, and we ignore 15 because the test generator cannot create tests for one of the two versions. For example, this may happen when the class contains a correctness bug that raises an exception. We consider all remaining 113 pairs of classes, of which we manually classify 15 as performance-influencing. Table 2 gives details for all class pairs with performance-influencing changes. The "Baseline" column of Table 2 shows which kind of performance difference we expect based on the manual inspection.

For each pair of classes, we run SpeedGun with 8 and 64 concurrent threads. In total, it generates 996 tests for the 113 pairs of classes. On average, there are 10 method calls in a test prefix and 111 method calls in a method suffix. We use the following parameters and thresholds. The warm-up phase t_w is 10 seconds; the steady-state phase is 20 seconds for 8 threads and 40 seconds for 64 threads. Based on initial experiments, we assume that the minimal measurable time span t_{min} is five milliseconds. For determining the test length, we use $\sigma_{stop} = 0.01$, $\sigma_{acceptable} = 0.02$, $m_{min} = 3$, $m_{max} = 5$, $maxTries = 20$, and $r_{min} = 50$. The test oracle uses $\delta_{report} = 0.05$ (Section 5.4 evaluates the influence of the threshold). To avoid biasing the results towards a particular platform, all experiments are done on two hardware platforms and with two different JVMs: (1) Intel Core i7 CPU (1.7 GHz, 8 cores) with Oracle JDK 1.7.0_13-b20 for 64-bit systems; (2) Intel Xeon CPU (2.53 GHz, 8 cores) with OpenJDK IcedTea7 2.1.3, running in a virtualized environment (VMWare vSphere). Unless noted otherwise, the results from both platforms coincide.

5.3 Reported Performance Differences

Applying the analysis to the 113 pairs of classes listed in Table 1 results in 13 reports about a performance difference. Based on these results, we can distinguish four cases.

Table 2: Changes and how they influence the concurrent performance of the changed class. The arrows indicate the performance influence of a change according to the baseline and to SpeedGun, where ↘ means performance regression, ↗ means performance improvement, and → means no performance difference. The last columns show the speedup measured by SpeedGun (averaged over all tests executed on the Intel Core i7 platform). For all pairs of classes that are not listed, we neither expect a performance difference nor does SpeedGun report any.

ID	Code base	Class	Revision	Description	Performance		
					Baseline	SpeedGun	
						8 thr.	64 thr.
(1)	Pool	GenericObjectPool	774007	Finer-grained locking to avoid deadlocks described in Issue 125	↗	↗ 1.52	↗ 1.57
(2)	Pool	GenericObjectPool	603449	Replace synchronized methods with volatile fields to address Issue 113	↗	↗ 1.30	↗ 1.38
(3)	Pool	GenericObjectPool	602773	Fix of a performance problem (Issue 93) by introducing more fine-grained locking	↗	↗ 2.09	↗ 2.20
(4)	Collections	StaticBucketMap	1076039	Fix of a correctness bug (Issue 334) by adding synchronization	↘	↘ 0.64	↘ 0.61
(5)	JodaTime	DateTime	v2.1	Newer version is reported to decrease performance over v1.5.2 due to additional synchronization (Issue 153)	↘	↘ 0.91	↘ 0.91
(6)	Groovy	ExpandoMetaClass	d3da3a44	Add synchronized blocks to fix correctness problem	↘	↘ 0.48	↘ 0.52
(7)	Groovy	ExpandoMetaClass	1c947d6b	Replace synchronized collections with project-internal concurrent collections	↗	↗ 1.08	↗ 1.29
(8)	Groovy	ExpandoMetaClass	2b09801e	Add synchronized block to fix correctness problem	↘	→	→
(9)	Groovy	ExpandoMetaClass	feff5190	Synchronize methods to fix correctness bug (Issue 2166)	↘	↘ 0.92	↘ 0.96
(10)	Groovy	ExpandoMetaClass	83629dc1	Patch to improve (sequential) performance	→	↗ 1.39	↗ 1.38
(11)	Groovy	ExpandoMetaClass	77822d4c	Replace project-internal concurrent collections with java.util.concurrent collections	↗	→	→
(12)	Groovy	ExpandoMetaClass	6e349cd9	Large patch without any obvious effects on performance	→	↘ 0.88	↘ 0.91
(13)	Groovy	ExpandoMetaClass	26fc2100	Replace synchronized methods with volatile field to fix performance bug (Issue 3557)	↗	↗ 1.50	↗ 1.42
(14)	Groovy	ExpandoMetaClass	d92c12ab	Replace volatile fields with synchronized methods to fix correctness problem	↘	↘ 0.95	↘ 0.95
(15)	Groovy	ExpandoMetaClass	48269129	Replace synchronized method with volatile fields to address performance problem (Issue 4182)	↗	↗ 1.03	→
(16)	Groovy	ExpandoMetaClass	cdc39843	Supposed performance improvement by replacing synchronized method with explicit locks	↗	→	→
(17)	Groovy	ExpandoMetaClass	d38da33c	Replace volatile field with synchronized method to fix correctness bug	↘	→	→

Neither expected nor reported a difference.

For 96 pairs of classes (85%), we do neither expect a performance difference, nor does SpeedGun report any. That is, for most changes of a class, the analysis does not bother developers with unnecessary reports.

Report of expected performance difference.

For eleven pairs of classes, SpeedGun reports a performance difference that we expect based on our manual inspection. These reports include several performance regressions that are apparently unintended side effects of fixing correctness problems. For example, the analysis reports a performance regression for Change (9) in Table 2, which is the regression from Figure 1(b). By reporting the performance regression right after applying the change, SpeedGun could have helped the developers to avoid this apparently unintended performance degradation. The analysis reports a performance improvement for Change (13), which is the improvement from Figure 1(c). For this change, SpeedGun could have helped the developers to check whether the supposed optimization works as intended. For change (15), SpeedGun identifies a small performance difference when testing with 8 threads but cannot measure any difference when testing with 64 threads.

Expected performance difference not reported.

For four pairs of classes, SpeedGun does not report a performance difference even though we expect it based on our initial manual inspection. Close inspection reveals that the changes indeed do not significantly influence the CUT's per-

formance:

- Change (8) fixes a thread safety bug by adding a small synchronized block that uses the instance of `ExpandoMetaClass` as the lock. SpeedGun does not report any performance difference because the change does not reduce the overall performance of the CUT in a significant way.

- After change (11), the CUT uses concurrent collections from `java.util.concurrent` instead of project-specific concurrent collection classes. The commit message is "Faster class info access", suggesting that the developers expect a performance improvement. However, SpeedGun does not confirm this expectation.

- Change (16) addresses a reported performance problem by replacing synchronized methods with explicit read and write locks. Although the developers seem to expect a performance improvement, SpeedGun does not confirm this expectation.

- Change (17) makes two methods synchronized, which (at first sight) seems to reduce performance. The reason why SpeedGun does not report a performance difference is that all call sites of the two methods are in other synchronized methods. That is, the change does not add any locking and therefore, there is indeed no performance difference.

These four examples illustrate that SpeedGun can help in verifying whether an expected performance difference is measurable in practice. Traditionally, developers manually develop micro-benchmarks for this purpose. Our analysis relieves developers from this cumbersome task and supports them in accurately measuring performance.

Report of unexpected performance difference.

For two pairs of classes, SpeedGun reports a performance difference that we do not expect based on our initial manual analysis:

- Change (10) is a relatively large change entitled "Performance patch". The patch introduces several optimizations that improve the sequential performance of the CUT. Although none of the changes is directly related to synchronizing concurrent threads, improving the sequential performance also influences the concurrent performance, which is why our analysis reports an improvement.

- Similar to Change (10), Change (12) is a relatively large patch. According to the commit message it fixes several correctness bugs. Based on our limited knowledge of the CUT, we suspect that the change degrades the sequential performance and therefore also the concurrent performance of the CUT.

These two examples illustrate that, even though SpeedGun aims at finding differences in the concurrent performance of a CUT, it may also report differences that are not directly related to concurrency. Developers that apply optimizations, such as Change (10), will not be surprised if an analysis reports a performance improvement. Therefore, we consider this behavior of the analysis to be acceptable.

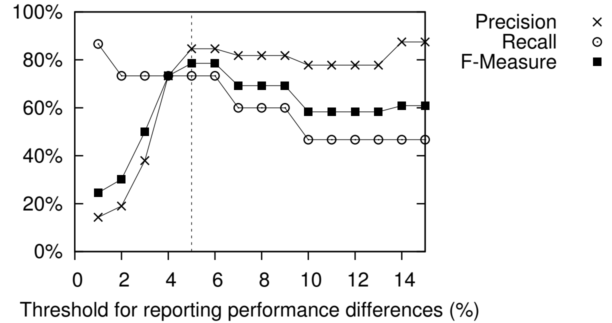

Figure 3: Sensitivity of reported warnings to the threshold δ_{report} for reporting warnings. The dashed vertical line indicates the default configuration.

5.4 Sensitivity to Threshold

SpeedGun decides when to report a performance difference based on a threshold δ_{report}. Figure 3 shows how this threshold influences the precision and the recall of the analysis. Precision is the number of reported warnings that are expected according to Table 2 divided by the total number of warnings. Recall is the number of warnings detected as expected divided by the total number of expected warnings. The figure shows that with a small threshold, the analysis detects more relevant warnings but also produces many more irrelevant warnings than with a large threshold. Our default configuration is $\delta_{report} = 5\%$, which is a pragmatic compromise between reporting too many and too few warnings. The results in Figure 3 are from the Intel Core i7 platform; the Intel Xeon platform gives similar results.

5.5 Performance of the Analysis

The running time of SpeedGun ranges between one hour and six hours per pair of classes. Most of the time is spent in repeatedly executing generated tests, which is necessary to accurately measure performance. Since the analysis is fully automated, we consider this running time to be acceptable.

6. LIMITATIONS

There are several limitations to be addressed in future work. First, the test generator may create tests that do not represent realistic usage scenarios, and therefore report performance differences that are not relevant in practice. To address this limitation, one may leverage real-world usage traces [5] or existing code that uses the CUT [50] to focus test generation on realistic scenarios. Second, the test generator may fail to create a test that triggers a particular performance problem and therefore miss a performance difference, for example, because triggering the problem requires a specific setup. Improvements in test generation may mitigate this limitation, but non-exhaustive testing will always risk to miss problems. Third, SpeedGun does not distinguish concurrency-related performance changes from sequential performance changes. One way to address this limitation is to apply the approach only to changes that are known to influence the concurrent behavior of a CUT. As an alternative, one could generate sequential tests to find sequential performance differences, and leverage this knowledge to mark them as unrelated to concurrency.

7. RELATED WORK

7.1 Performance Analysis

Existing dynamic analyses find performance problems related to excessive memory usage [55, 54], repeated computations [38], and latency in GUI applications [31]. Grechanik et al. describe a dynamic analysis that selects a subset of given inputs to find performance problems [22]. StackMine analyzes call stack traces of many program executions to automatically detect likely performance problems [23]. In contrast to these approaches, which use manually created input to exercise the program under test, SpeedGun is driven by automatically generated tests.

Other work focuses on understanding performance problems, for example, by profiling [27, 24, 1] and by systematically narrowing down the set of potential root causes [52]. Attariyan et al. tracks a performance problem back to a part of the input of the program by comparing different inputs [3]; instead, our approach compares different versions of a class with the same input. PerfDiff compares executions of a program on different platforms to find and localize performance problems [60]. Instead, SpeedGun compares multiple versions of a program on a single platform.

Studies of performance bugs show that SpeedGun addresses a relevant problem: unnecessary synchronization that intensifies thread competition commonly causes performance bugs [28], 22% of all performance bugs are regressions [58], and 14% of all performance bugs are release-blocking [58].

7.2 Regression Testing

McCamant and Ernst propose to detect correctness problems that result from changes by comparing abstractions of dynamically observed behavior [34]. Jin et al. propose to compare two versions of a class by dynamically comparing their behavior and by presenting the user a ranked list of observed behavioral differences [29]. Their work and our work share the idea of generating unit-level tests for regression testing. In contrast to both [34] and [29], SpeedGun targets performance problems. McKeeman proposes to test supposedly equivalent programs, such as multiple compilers for the same language, by comparing them with each other [35]. This idea, called differential testing, has also been used to test system programs [9] and refactoring engines [14]. In contrast to differential testing, SpeedGun compares two versions of the same program.

Some large projects have manually written performance regression tests [56, 10]. Foo et al. propose to identify performance problems exposed by such tests by analyzing correlations between performance metrics [18]. Our work improves upon these approaches by creating tests automatically.

7.3 Concurrency Bugs

Various approaches find concurrency-related correctness bugs, such as data races [16, 44], atomicity violations [2, 17, 46], and violations of inferred invariants [33, 47]. Our own previous work generates multi-threaded unit tests and compares their concurrent execution to linearizations [41]. These analyses target correctness problems and therefore complement SpeedGun, which targets performance problems.

Another line of research to help detecting concurrency bugs are approaches to influence the scheduler by adding random delays [15], by exploring interleavings systematically [36, 12] or randomly [7], by forcing schedules that expose potential concurrency bugs [45, 40, 32, 30, 26], and by forcing schedules that cover yet uncovered interleavings [25, 57, 49]. By influencing the scheduler, these approaches also influence the performance of a concurrent program, and therefore are not directly applicable in our approach.

7.4 Test Generation

Various techniques to generate sequential tests have been proposed, such as random test generation [13, 39, 11], techniques based on model checking [51], and techniques based on symbolic execution [20, 53, 9]. Recent work extends these ideas to multi-threaded test generation [41, 37, 48]. In contrast to these existing approaches, SpeedGun addresses the challenges of generating concurrent performance tests. Zhang et al. propose mixed symbolic-concrete test generation that focuses on paths that are estimated to have a high performance impact [59]. Wise generates tests that expose the worst-case complexity of a program [8]. Both approaches do not address concurrent programs. Avritzer and Weyuker describe algorithms to create tests for concurrent telecommunication systems [4]. Their approach is limited to systems that can be modeled as Markov chains.

SpeedGun is part of a line of work that combines dynamic analyses with test generation targeted at a particular analysis [43, 42, 41]. In contrast to the existing approaches, SpeedGun targets performance problems.

8. CONCLUSIONS

This paper presents SpeedGun, an automatic approach to detect performance differences between versions of thread-safe classes. The approach is enabled by combining a generator of concurrent performance tests and a component that addresses the challenges of evaluating concurrent performance on a virtual machine. To validate the practicality of SpeedGun, we evaluate its effectiveness on a collection of 113 pairs of real-world Java classes. The evaluation shows that the analysis effectively identifies performance differences between versions of these classes, including performance regressions that the developers were not aware of. Although the implementation targets Java classes, the result is not Java-specific and we expect the approach to work for other object-oriented languages.

The analysis presented here fills the gap between existing techniques targeted at correctness of concurrent programs and traditional approaches to assess and enhance their performance, such as profiling and manually written microbenchmarks. By filling this gap with an automatic analysis, SpeedGun is the foundation of a practical tool that brings developers a step closer towards meeting the conflicting goals of correctness and performance of concurrent programs.

Our implementation and artifacts to reproduce our results are available at `http://mp.binaervarianz.de/SpeedGun/`.

9. ACKNOWLEDGMENTS

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453. We thank Cindy Rubio González, Joel Galenson, and the anonymous reviewers for their comments on this paper.

10. REFERENCES

[1] E. R. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753. ACM, 2010.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[3] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2012.

[4] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, 1995.

[5] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *Conference on Automated Software Engineering (ASE)*, pages 333–342, 2007.

[6] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340. ACM, 2010.

[7] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[8] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX, 2008.

[10] T. Chen, L. I. Ananiev, and A. V. Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, 2007.

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*, pages 71–80. ACM, 2008.

[12] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pages 15–24. ACM, 2010.

[13] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, 2004.

[14] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 185–194. ACM, 2007.

[15] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM, 2009.

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM, 2008.

[18] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *International Conference on Quality Software (QSIC)*, pages 32–41, 2010.

[19] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, pages 57–76. ACM, 2007.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[21] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[22] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *International Conference on Software Engineering (ICSE)*, pages 156–166, 2012.

[23] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155. IEEE, 2012.

[24] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 251–269, 2004.

[25] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 210–220. ACM, 2012.

[26] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 144–154. ACM, 2011.

[27] M. Ji, E. W. Felten, and K. Li. Performance measurements for multithreaded programs. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 161–170, 1998.

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[29] W. Jin, A. Orso, and T. Xie. Automated behavioral

regression testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 137–146. IEEE, 2010.

[30] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 110–120. ACM, 2009.

[31] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–170. ACM, 2011.

[32] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering (ICSE)*, pages 235–244. ACM, 2010.

[33] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Symposium on Microarchitecture (MICRO)*, pages 553–563. ACM, 2009.

[34] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 287–296. ACM, 2003.

[35] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[36] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, 2008.

[37] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *International Conference on Software Engineering (ICSE)*, pages 727–737, 2012.

[38] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE, 2007.

[40] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Symposium on Foundations of Software Engineering (FSE)*, pages 135–145. ACM, 2008.

[41] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[42] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering (ICSE)*, pages 288–298, 2012.

[43] M. Pradel and T. R. Gross. Automatic testing of sequential and concurrent substitutability. In *International Conference on Software Engineering (ICSE)*, pages 282–291, 2013.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[45] K. Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21. ACM, 2008.

[46] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 51–64, 2011.

[47] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 160–174. ACM, 2010.

[48] S. Steenbuck and G. Fraser. Generating unit tests for concurrent classes. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[49] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Conference on Automated Software Engineering (ASE)*, 2013.

[50] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 193–202. ACM, 2009.

[51] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.

[52] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *International Conference on Software Engineering (ICSE)*, pages 552–561, 2013.

[53] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381. Springer, 2005.

[54] G. Xu. Finding reusable data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034. ACM, 2012.

[55] D. Yan, G. H. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering, (ICSE)*, pages 134–144. IEEE, 2012.

[56] C. Yilmaz, A. S. Krishna, A. M. Memon, A. A. Porter, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Main effects screening: a distributed

continuous quality assurance process for monitoring performance degradation in evolving software systems. In *International Conference on Software Engineering (ICSE)*, pages 293–302, 2005.

[57] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 485–502. ACM, 2012.

[58] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Working Conference on Mining Software Repositories (MSR)*, pages 199–208. IEEE, 2012.

[59] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Conference on Automated Software Engineering (ASE)*, pages 43–52, 2011.

[60] X. Zhuang, S. Kim, M. J. Serrano, and J.-D. Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Symposium on Code Generation and Optimization (CGO)*, pages 4–13. ACM, 2008.