BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code

Adrian Nistor[1], Qingzhou Luo[1], Michael Pradel[2], Thomas R. Gross[2], Darko Marinov[1]
[1] *Department of Computer Science, University of Illinois at Urbana-Champaign, USA*
[2] *Department of Computer Science, ETH Zurich, Switzerland*
{*nistor1, qluo2, marinov*}*@illinois.edu, michael@binaervarianz.de, thomas.gross@inf.ethz.ch*

***Abstract*—Testing multithreaded code is hard and expensive. A multithreaded unit test creates two or more threads, each executing one or more methods on shared objects of the class under test. Such unit tests can be generated at random, but basic random generation produces tests that are either slow or do not trigger concurrency bugs. Worse, such tests have many false alarms, which require human effort to filter out.**

We present BALLERINA, a novel technique for automated random generation of efficient multithreaded tests that effectively trigger concurrency bugs. BALLERINA makes tests efficient by having only two threads, each executing a single, randomly selected method. BALLERINA increases chances that such simple parallel code finds bugs by appending it to more complex, randomly generated sequential code. We also propose a clustering technique to reduce the manual effort in inspecting failures of automatically generated multithreaded tests. We evaluate BALLERINA on 14 real-world bugs from six popular codebases: Groovy, JDK, JFreeChart, Apache Log4j, Apache Lucene, and Apache Pool. The experiments show that tests generated by BALLERINA find bugs on average 2X-10X faster than basic random generation, and our clustering technique reduces the number of inspected failures on average 4X-8X. Using BALLERINA, we found three previously unknown bugs, two of which were already confirmed and fixed.

I. INTRODUCTION

General-purpose processors manufactured today have multiple cores, and the projections are that the number of cores will be increasing. To harness these cores for speeding up their applications, developers now write parallel code, typically using a multithreaded, shared-memory programming style. However, writing multithreaded software is notoriously difficult, as the same code can have different behavior for different thread interleavings. Additionally, testing multithreaded software is expensive because it requires (1) writing appropriate test code, (2) exploring (many) thread interleavings of this test code while checking oracles, and (3) inspecting oracle violations.

Most research on testing multithreaded software [1]–[6] has focused on improving approaches for step (2)—exploring thread interleavings and checking generic oracles. These approaches typically do not generate test code but only explore the given test code for various thread interleavings and apply generic oracles checking for bugs such as data races, atomicity violations, non-determinism, or non-linearizability [2], [7]–[12]. If an interleaving violates an oracle, a potential bug is reported to the developer. These approaches have shown a lot of promise in finding real bugs. However, they require to be given test code as input, and they can produce a lot of false alarms.

Automatically generating unit test code for multithreaded software could greatly ease testing of such software. While many techniques have been proposed for automated generation of test code for sequential software, e.g., random generation of unit tests for object-oriented software [13]–[15], they do not directly apply to multithreaded software. Namely, test code for multithreaded, object-oriented software needs to create one or more objects that are *shared* among multiple threads and to invoke methods on these objects; putting arbitrary test code for sequential software into multiple threads rarely creates enough sharing to trigger concurrency bugs.

A *basic random generation* of multithreaded tests [2] can achieve sharing by first creating an object sequentially using a randomly selected *constructor* and then invoking in *several parallel threads* randomly selected methods on this object. Unfortunately, this generation has two major problems. First, it can take a lot of machine time to explore the generated test code, because the total number of thread interleavings grows exponentially with the number of threads and sharing among threads. Second, it can take a lot of developer time to inspect the reports generated by violations of generic oracles for multithreaded tests, because such oracles can create a lot of false alarms (e.g., on the order of ten false alarms to one true bug report, as shown in Section V).

This paper makes three contributions.

A novel technique for generating efficient multithreaded tests that are effective at triggering concurrency bugs: We propose BALLERINA, a novel technique for automated random generation of efficient multithreaded unit test code. BALLERINA takes as input a class under test and a set of its methods, and generates as output multithreaded tests that are *both* efficient and effective at triggering concurrency bugs. The key idea is to minimize parallel execution to reduce the time needed for exploring it, without sacrificing the bug-triggering capabilities. The minimal possible configuration for multithreaded unit tests

is *two threads that each execute a single method under test*. Combining this minimal configuration with basic random generation would miss many real-world concurrency bugs. Instead, BALLERINA prepends the minimal *parallel suffix* with a more complex *sequential prefix* that "grows" the object(s) under test, creating the conditions necessary to trigger bugs in the parallel suffix.

BALLERINA focuses on bugs that can be triggered by two threads. This suffices for almost all real-world concurrency bugs: a recent study [16] of 96 concurrency bugs in large C/C++ applications (MySQL, Apache, Mozilla, and OpenOffice) found that only two bugs require at least three threads; in other words, 94 out of 96 bugs can be reproduced with only two threads.

After BALLERINA generates multithreaded tests, it needs to explore and check them. Exploration is orthogonal to generation, so BALLERINA could use stress testing [3], randomized thread interleavings [17]–[20], and/or systematic exploration [3], [6], [21]. Our BALLERINA implementation uses Java PathFinder (JPF) [6]. For checking, our implementation uses *linearizability* [2], [22], which intuitively reports a violation whenever a thread interleaving produces a behavior that cannot be produced in any linearized test execution where all methods execute atomically. We chose linearizability because it can find real concurrency bugs [2].

Clustering oracle violations for faster inspection: While some linearizability violations show true bugs, other violations are false alarms. For example, the Apache Pool can throw `ConcurrentModificationException` in some concurrent scenarios but does not throw this exception in a corresponding sequential execution. This behavior is a linearizability violation, but it is clearly specified in the Pool documentation, so it is a benign violation, not a true bug. Ideally, a perfect oracle would not report this behavior, but all known generic oracles for multithreaded software create false alarms, e.g., 90% of reported data races are benign [23].

BALLERINA alleviates the problem of false alarms using *clustering*. We build on the idea of clustering failures for sequential code [24]–[29]: first, reports are split into clusters with similar failures that are likely to be either all false alarms or all true bugs; then, rather than inspecting the reports in an arbitrary order, the developer can inspect them by selecting from different clusters. BALLERINA performs clustering based on concurrently executing methods under test and the type of failure.

Evaluation: We evaluate BALLERINA on *14 real-world bugs* from six popular codebases: Groovy, JDK, JFreeChart, Apache Log4j, Apache Lucene, and Apache Pool. We compare BALLERINA with basic random generation of multithreaded tests of six different sizes (based on the number of threads and methods). We consider four different exploration approaches (CHESS-like preemption bounding [3], exhaustive stateful search used in JPF [6], stateless search [21], and parallelized test execution) and measure the cost of exploring generated tests to find a bug. Compared to basic random generation of minimal tests (two threads, each with one method), BALLERINA *finds more bugs* (13 vs. 3). Compared to basic random generation of larger tests (two or more threads, each with two or more methods), *tests generated by* BALLERINA *trigger the bugs substantially faster*, 2X-10X on average.

```
1  class GenericObjectPool extends BaseObjPool implements ObjPool {
2      GenericObjectPool(PoolableObjectFactory factory, int maxActive) {...}
3      Object borrowObject() {...}
4      void returnObject(Object obj) {...}
5      void addObject() {...}
6      void invalidateObject(Object obj) {...}
7      void clear() {...}
8      void evict() {...}
9      void close() {...}
10     void setMaxActive(int maxActive) {...}
11     int getMaxActive() {...}
12     ...
13 }
```

Figure 1. API for the `GenericObjectPool` class from Apache Pool

We evaluate the effectiveness of our clustering technique for both BALLERINA and the basic random generation. For some cases, there are no false alarms, i.e., all reported violations are true bugs. For some other cases, there can be a large number of false alarms. Without clustering, the developer would need to sift through a large number of reports (e.g., dozens) to find a true bug. *Our clustering reduces inspection time*: the developer needs to inspect a small number of reports (e.g., 3-4) to find a true bug.

Using BALLERINA, we found *three previously unknown bugs* in the widely used Apache Log4j and Apache Pool. Two bugs were confirmed and fixed [30], [31], and the third is still under investigation [32].

II. EXAMPLE

To illustrate how BALLERINA works, we use the example of testing a class from the Apache Pool library. The library provides several classes that implement pools, i.e., collections of objects that can be shared among several threads. Figure 1 shows declarations of several methods from the `GenericObjectPool` class. The constructor takes a `Factory` that creates objects for the pool, and sets the `maxActive` number of objects in this pool. Clients of this class first obtain an object from the pool (`borrowObject`), then work with this object, and finally return it back to the pool (`returnObject`). The methods `clear` and `evict` remove from the pool all idle objects (i.e., those not currently borrowed) and all objects that satisfy certain criteria, respectively. The class also has a large number of getter and setter methods, but we show only those for `maxActive`.

The `GenericObjectPool` class is highly concurrent. Because a number of clients can be invoking the pool methods from different threads, `GenericObjectPool` does not globally lock the entire pool for each method but rather

uses fine-grained locking so that a number of methods proceed concurrently.

Suppose that we want to test how `GenericObjectPool` behaves when clients call various methods from multiple threads. It is easy to apply BALLERINA to generate multithreaded unit test code for `GenericObjectPool`. We instruct BALLERINA to test this class and its methods listed in Figure 1. We also provide a simple factory class, because `PoolableObjectFactory` is an interface, and any test would need to provide a concrete class.

Generating Tests with BALLERINA: BALLERINA randomly generates tests, each of which consists of two parts. The *sequential prefix* creates an *object under test* (OUT), in our example a `GenericObjectPool` object, and invokes several methods on that object. The parameter values for the methods are randomly generated, by creating other helper objects as necessary. The *parallel suffix* creates two threads that each execute only one of the given methods on the OUT created in the sequential prefix. The test execution will explore multiple interleavings of these two threads.

Figure 2 shows an example test generated by BALLERINA. The OUT is `var4`. Its constructor sets `maxActive` to 1. After `var4` is created, `addObject` is invoked on it. Both the parameter value (1) and the method call (`addObject`) are required to bring `var4` in a state where a bug is triggered by the parallel suffix. In the parallel suffix, the threads `t1` and `t2` invoke `borrowObject` and `evict`, respectively, on `var4`. We also show `var2` generated by BALLERINA, although it is not necessary to trigger the bug in this case.

Executing the BALLERINA Tests: BALLERINA uses known techniques to explore different thread interleavings of generated tests [3], [6], [21] and to check for potential bugs using linearizability [2]. For example, exploring the test from Figure 2 with CHESS-like exploration (preemption bound of two) [3] finds a non-linearizability violation: executing `borrowObject` and `evict` atomically succeeds but executing them concurrently deadlocks. BALLERINA presents to the user a report with this test and its interleaving.

Real Unknown Bug: Our inspection of this example indeed revealed a previously unknown bug in `GenericObjectPool`, which the Pool developers confirmed and fixed after we had informed them. The analysis shows that triggering this bug requires a particular, non-trivial pool state and execution condition, as described by the Pool developers: "[...] whenever maxActive is about to be attained by a borrowObject with one idle instance in the pool, if one thread does a borrow while the evictor is visiting the idle instance, the borrowing thread will stall until another thread does a borrow or return." [30]. Such state cannot be created with a trivial sequential prefix, and having a more complex parallel suffix would increase the exploration time to find the bug.

Comparison with Basic Random Generation: Exploring many thread interleavings for a given test is expensive [4], [6]. The cost is increasing with the number of threads and the number of shared memory accesses in each thread. BALLERINA minimizes this cost by generating the minimal possible parallel section, two threads with one method each. In contrast, consider basic random tests that would have three or more threads and/or execute more than one method per thread. These tests would have more thread interleavings and would be slower to explore when they trigger no bug. However, it is not clear a priori that they would take more time to trigger the bug (because testing more methods at once could increase the chance to trigger the bug). In our example of `GenericObjectPool`, experimental results (Section V) show that BALLERINA tests trigger the bug about 5X faster than basic random tests with two threads and two methods each, while other configurations with more threads or methods are even slower.

```
1  void test() {
2      //sequential prefix, object under test (OUT): var4
3      final SimpleFactory var0 = new SimpleFactory();
4      final GenericObjectPool var2 = new GenericObjectPool(var0, 0);
5      final GenericObjectPool var4 = new GenericObjectPool(var0, 1);
6      var4.addObject();
7      //parallel suffix
8      Thread t1 = new Thread() {
9              public void run() { var4.borrowObject(); } };
10     Thread t2 = new Thread() {
11             public void run() { var4.evict(); } };
12     t1.start(); t2.start();
13     t1.join(); t2.join();
14 }
```

Figure 2. A test generated by BALLERINA that triggered an unknown bug

Report Clustering: To reduce the time for inspecting reports, BALLERINA introduces a novel report clustering based on the methods under test that execute at the point of failure and the type of failure. The intuition is that similar reports are likely to be either all false alarms or all true bugs. Our evaluation shows that, for `GenericObjectPool`, both BALLERINA and basic random tests can generate up to hundreds of false alarms for each true bug, which would require that the developer inspects a large number of false alarm reports to find a true bug, rendering the tool impractical. With our clustering technique, however, the developer needs to inspect only three or four false alarms before finding a true bug report.

III. GENERATING TESTS

We next describe how BALLERINA generates multithreaded unit tests for a given class under test (CUT) and a set of methods. Each test consists of two parts: a *sequential prefix* that creates an *object under test* (OUT) with a CUT constructor and potentially calls some methods on the OUT sequentially, and a *parallel suffix* that calls two methods on the OUT concurrently.

A. Generating Sequential Prefix

BALLERINA modifies the Randoop algorithm [14] to generate the OUT in the sequential prefix. In general, Randoop

takes as input a set of classes under test and generates sequential tests that have random sequences of method calls to these classes. Randoop *broadly but sparsely* covers states of objects from these classes. Even when given only one CUT, Randoop sparsely covers states of objects of that class, e.g., calling methods on many objects but not necessarily calling many methods on one object. In contrast, we want BALLERINA to *more densely* cover states of objects for the one given CUT. To that end, we modified Randoop to focus the generation on *one OUT* for each sequence and to "grow" such objects across various sequences.

Figure 3 shows the pseudo-code for the relevant part of the Randoop algorithm (adapted from [14]) and the BALLERINA addition highlighted (lines 8, 11, 12, 13). The algorithm maintains a collection of method sequences. For each sequence that can create one or more objects of the CUT, BALLERINA tags *exactly one* OUT. Some sequences create no objects of the CUT but only objects of other classes, such as `SimpleFactory` in the example from Figure 2. In that case, there is no OUT, but such sequences are still useful, because they can be used as parameters for methods in sequences that do have an OUT.

In lines 6 and 7, Randoop randomly chooses a method m (whose parameters have types τ_i), some sequences $seqs$ that can create method parameters, and expressions e_i for these parameters (e.g., a variable such as `var0`, or `null` if no object of a type is available, or constants for primitive types). In line 9, Randoop generates a new random sequence $newSeq$ by appending sequences for parameters and adding a new call $newVar = m(e_1 \dots e_n)$; it then checks that the new sequence does not fail with uncaught exceptions.

BALLERINA additionally selects and/or updates the OUT. In line 8, if the receiver has the CUT type, then BALLERINA does not randomly select the expression for that parameter but *selects the OUT from the appropriate sequence*. Thus, the OUT from the sequence $seqs(1)$ is enhanced by calling the method $m(\tau_1 \dots \tau_n)$ on it, and the same object becomes the OUT for the newly created sequence $newSeq$ in line 11. In line 12, BALLERINA creates a new OUT when m is a CUT constructor or a static method returning objects of the CUT type. If a method both returns an object of the CUT and has the receiver of the CUT type, then the OUT is the receiver. Effectively, this favors enhancing the state of OUT, as opposed to creating a new OUT.

While BALLERINA aims to create diversity of states for objects of the CUT, it also avoids redundant states that would only increase testing time but not the chance to find bugs. In line 13, BALLERINA checks if the new OUT was already generated by another sequence. Randoop has a similar check but for the objects produced by the last method call in the new sequence, while BALLERINA focuses on the OUT.

To illustrate, recall Figure 2 and consider the sequence that consists of the first three (constructor) calls—call it S. Assume that `var4` is the OUT for S and further

```
1  Algorithm RandomlyGenerateMethodSequences
2    input: classUnderTest (CUT), methodsUnderTest
3    output: collection of non-error method sequences
4  nonErrorSeqs ← emptyCollection
5  while (time limit not reached)
6    m(τ1 ... τn) ← randomlySelectOneMethod(methodsUnderTest)
7    ⟨seqs, e1 ... en⟩ ← randomSeqsAndExprs(nonErrorSeqs, τ1 ... τn)
8    if (τ1 = CUT) e1 ← seqs(1).OUT
9    newSeq, newVar ← append(seqs, m, e1 ... en)
10   if (executing newSeq fails) continue
11   if (τ1 = CUT) newSeq.OUT ← e1
12   else if (returnType(m) = CUT) newSeq.OUT ← newSeq.newVar
13   if (∃ s ∈ nonErrorSeqs s.t. s.OUT.equals(newSeq.OUT)) continue
14   nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
```

Figure 3. Integrating BALLERINA's generation of sequential prefix in the Randoop algorithm. BALLERINA code is highlighted.

that BALLERINA randomly selects the method `addObject`. The only parameter (i.e., the receiver) is of the type `GenericObjectPool`, which is the CUT. If BALLERINA selects the sequence S for that type, it then selects the OUT `var4` (rather than `var2`) as the expression for that parameter.

B. Generating Parallel Suffix

After generating a collection of sequences, BALLERINA uses them to generate multithreaded tests by adding code for the threads. For a parallel suffix, BALLERINA first randomly selects two methods from the given set. The selection is not uniform but based on the number of method parameters of the CUT type. We call these parameters *CUT-parameters*. A method that has two or more CUT-parameters is twice as likely to be selected than a method that has one or zero CUT-parameters. The intuition for this bias is that methods with more CUT-parameters have more interactions between the objects of the CUT and are thus more likely to trigger concurrency bugs. Of the methods in Figure 1, none has more than one CUT-parameter, so they are all equally likely to be selected; for the example in Figure 2, BALLERINA selected `borrowObject` and `evict`.

After selecting two methods, BALLERINA randomly selects parameters for these methods. BALLERINA first randomly selects from the collection one or two sequences with an OUT to use in the sequential prefix. BALLERINA uses two sequences if any of the two selected methods has two or more CUT-parameters; otherwise, it uses one sequence. We call these sequences *selected sequences*. For our running example with `borrowObject` and `evict`, there is one selected sequence, which ends with `var4.addObject`.

BALLERINA next selects the expressions for the parameters of the two methods. If a parameter type is the CUT, BALLERINA randomly selects the OUT from one of the selected sequences. Otherwise, BALLERINA randomly selects an object of the appropriate type from the selected sequences (because those sequences can have other objects besides the OUT) or, if there is no such object, BALLERINA selects a sequence from the entire collection. In our example, each (receiver) parameter for `borrowObject` and `evict` is

from the CUT, and thus BALLERINA selects `var4`, resulting in the test shown in Figure 2.

IV. REPORT CLUSTERING

BALLERINA uses linearizability [2], [22] as a generic oracle and may report false alarms. To alleviate the problem of inspecting failure reports, BALLERINA can cluster test failures based on their similarity. Each failure report contains several pieces of information: (1) the test that was executed, including the methods called from the test (BALLERINA generates only two methods per test, but the basic random generation can generate a larger number of methods per test), (2) the thread interleaving that was executed up to the failure, (3) the stack trace for each thread, including the *executing methods* (i.e., the methods under test that were executing at the point of the failure), and (4) the *type of failure* (deadlock or an exception, including the class of the exception, e.g., `NullPointerException` or `ConcurrentModificationException`). In general, all these pieces could be used to determine failure similarity.

BALLERINA uses only the executing methods and the type of failure to determine similarity. Our experiments show that these two pieces of information already provide excellent results for clustering failures of automatically generated multithreaded unit tests. More specifically, BALLERINA splits the failures into clusters such that all failures in one cluster have (1) the same set of executing methods under test (i.e., the allocation of methods to threads does not matter) and (2) the same type of failure. Note that several reports from the same test can end up in different clusters, and reports from different tests can end up in the same cluster. Indeed, while (deterministic) sequential tests can have only one outcome (pass or fail), a multithreaded test can have different outcomes for different thread interleavings, some of which may be false alarms while others are true bugs.

Splitting reports into clusters is the first step in using clustering; the next step is determining the *sampling strategy* [24], [27] for inspecting reports from the clusters. A common sampling strategy [24], [27], which we also use, is to randomly order clusters and visit them in a round-robin fashion, randomly selecting one report for inspection from each cluster. Another strategy, specific to the multithreaded code, would be to inspect the failures in a cluster in the order in which the test exploration produced them. We found that the latter strategy does not provide better results than the former because two consecutive failures have similar thread interleavings and thus are likely to be either both false alarms or both true bugs.

V. EVALUATION

We evaluate BALLERINA on *14 real-world bugs* from six popular codebases: Groovy, JDK, JFreeChart, Apache Log4j, Apache Lucene, and Apache Pool. Figure 4 shows the following information about each bug: the application name, the bug ID that we will use in the rest of the paper, the issue ID from the application's issue-tracking system, the number of lines of code in the application, the class under test, its number of lines of code, the number of methods we give BALLERINA to test, and the type of bug. We chose the methods to test among the more complex methods in the CUT, simulating how an expert developer of multithreaded code would use her intuition to focus the BALLERINA tool.

Our evaluation addresses these three research questions:

RQ1: Do tests generated by BALLERINA find more bugs and/or find bugs faster than tests generated by basic random generation? A basic random test has T threads, each executing M methods under test; it is not obvious which $T \times M$ configuration works the best, so we evaluate six: 2×1, 2×2, 2×3, 3×1, 3×2, and 3×3. For basic random, the sequential prefix randomly chooses a constructor (and its parameter objects) for the OUT, while BALLERINA also has up to three method calls on the OUT; both potentially have more method calls for the other objects used as parameters.

RQ2: Does the speedup provided by BALLERINA vary for different exploration approaches? We consider CHESS-like preemption bounding [3], exhaustive stateful search used in JPF [6], stateless search [21], and parallelized test execution. The reason for considering multiple exploration approaches is that there is no established best approach.

RQ3: Does our clustering technique reduce the effort of inspecting violations reported by exploration of tests generated by BALLERINA and basic random generation?

To compare the exploration cost of tests, we use the number of transitions executed by stateful JPF while exploring the state space and the number of execution paths for the stateless exploration based on re-execution. This is consistent with previous studies on exploration costs [17], [34], [35]. We do not use the actual real time from JPF because we conducted the experiments on a computing cluster that has machines with different hardware. Note that we compare cost to *explore not to generate* tests, because generation cost is about the same for various techniques and often much smaller than exploration cost.

Both BALLERINA and basic random generate tests using random selection. To check how the results vary for different random seeds, we evaluate both BALLERINA and basic random using 200 seeds.

A. Answering RQ1

Figure 5 shows the cost that CHESS-like preemption-bounded exploration [3] incurs to find a true bug for tests generated by BALLERINA and basic random. For each bug we show seven box plots, one for BALLERINA (BLR) and six for configurations of basic random generation. Each box plot shows five values that summarize the exploration cost over 200 random seeds: median, upper and lower quartile values, and the max and min values not outside the 1.5 inter-quartile range. All the values are normalized to 1.0 for the

Application	BugID	Issue ID	Total LOC	Class Under Test (CUT)	CUT LOC	# Methods	Error
Groovy	#1	1890	54,872	MemoryAwareConcurrentReadMap	315	5	Deadlock
JDK	#2	[33]	335	StringBuffer	335	9	Deadlock
JDK	#3	4779253	2,555	Logger	523	11	Exception
JDK	#4	6487638	2,550	Logger	527	9	Deadlock
JFreeChart	#5	806667	65,027	NumberAxis	1,059	7	Exception
Log4j	#6	509	10,273	FileAppender	185	13	Exception
Log4j	#7	1507	10,770	Category	387	9	Exception
Log4j	#8	26224	20,098	AsyncAppender	171	12	Deadlock
Log4j	#9	38137	15,875	AsyncAppender	161	12	Deadlock
Log4j	#10	51783	21,033	WriterAppender	171	12	Exception
Lucene	#11	1358	57,347	PhraseQuery	184	7	Deadlock
Pool	#12	146	12,615	GenericKeyedObjectPool	1,049	10	Deadlock
Pool	#13	149	11,337	GenericObjectPool	782	9	Deadlock
Pool	#14	184	11,880	GenericObjectPool	737	9	Starvation

Figure 4. Basic statistics about the tested code and bugs. Issue ID is from the respective bug database.

BALLERINA median for the respective bug. For example, for Pool#14 (a new bug we found in Pool), the median of 2×2 is about 5, which means it is about 5X *slower* (*not faster*) than the median of BLR. Note that some values are out of bound, in which case we show *the smallest* of the five values that is out of bound. For example, for Pool#14 and 2×2, the next value that is missing is upper quartile (and it is 16). In some cases a technique does not find the bug, which is marked with 'not found' (but we still show the exploration cost for the generated tests). For example, for Pool#14, the tests generated by 2×1 do not find the bug.

The tests generated by 2×1 miss eleven out 14 bugs, while tests generated by 3×1 miss five out of 14 bugs. In contrast, BALLERINA and all other configurations miss only one out of 14 bugs (JDK#4). Triggering this bug requires that the object state has a special string format that is unlikely to be created by any random generation. More specifically, triggering the bug requires a call to the `readConfiguration(InputStream ins)` method; the `ins` parameter can come from a string, but it needs to be in the `java.util.Properties` format, as specified by the `readConfiguration` documentation. Thus, using a random string like `"Hi!"` does not suffice, and Randoop does not randomly generate strings like `"x.level=FINE"`.

> *Tests generated by* BALLERINA *can find more bugs than tests generated by simple basic random configurations (2×1 and 3×1).*

We next turn to comparing the costs for the basic random configurations that find the same bugs as BALLERINA. Figure 5 shows the distribution of costs over random seeds. We can see that the cost for BALLERINA is often lower than the cost for basic random, although there are individual scenarios when basic random is faster than BALLERINA, e.g., for Pool#14, the min whisker of 2×2 is close to 0, whereas the max whisker of BALLERINA is close to 4. To summarize the comparison into one number, we compute the *arithmetic mean* cost across random seeds (whereas box plots show *median* values).

Figure 6 shows the mean exploration cost for all bugs and compared configurations. For now we discuss the left half of the table, for CHESS-like preemption bounding. We tabulate the mean number of transitions that exploration of tests generated by BALLERINA take to find the bug, and the slowdown of basic random configurations (computed as the ratio of their mean over the mean for BALLERINA). The last row shows the average slowdown, computed as the *geometric mean*. The cells marked with 'n/f' represent cases when a configuration does not find a bug.

For the basic random tests with two threads, the configuration that finds the bugs fastest on average is 2×3, and *not* 2×2 as one might expect. The reason is that 2×3 executes six methods per test, whereas 2×2 executes only four, and while exploration of a given 2×2 test is faster than a given 2×3 test, 2×3 has a higher chance of finding a bug. The individual slowdown of these configurations over BALLERINA ranges from 1.3X (for JDK#2 and 2×2) to 11.1X (for JDK#3 and 2×2). For two cases, Groovy#1 and Log4j#6, 2×2 and 2×3 even find a bug faster than or as fast as BALLERINA (and their speedup is reflected as "slowdowns" of less than or equal to 1.0). Triggering these bugs does not require complex object state, just parallel calls to the buggy pair of methods. In fact, these are two bugs that even 2×1 can find, because they are so simple to trigger.

The basic random tests with three threads are quite slow, with average slowdown of over 8X compared to BALLERINA. Considering the large cost of tests with three threads and the recent study [16] which found that only two of the 96 real-world concurrency bugs in large C/C++ applications require at least three threads to trigger, we believe that focusing BALLERINA on two threads is the right trade-off. If the programmer wants to look for bugs with three or more threads, she would first run fast tests with two threads and only later run the slower tests with three threads.

> *Tests generated by* BALLERINA *find bugs faster than tests generated by basic random configurations (which find the same bugs), on average 2.6X-10.4X.*

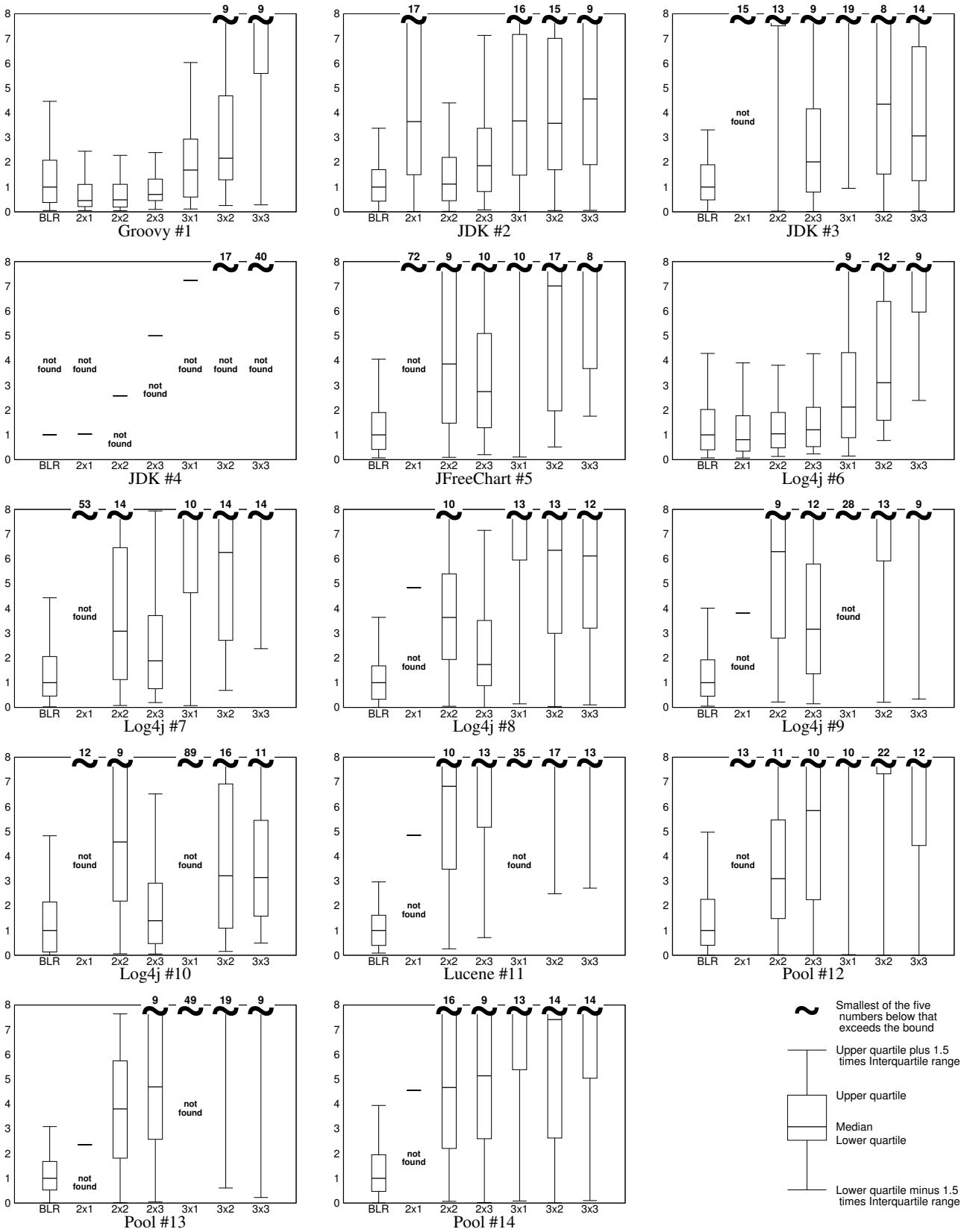

Figure 5. Number of transitions to the first execution for a given bug scenario

Bug	Preemption-bounded exploration							Exhaustive JPF exploration						
	BLR (absolute)	Slowdown relative to BLR						BLR (absolute)	Slowdown relative to BLR					
		2x1	2x2	2x3	3x1	3x2	3x3		2x1	2x2	2x3	3x1	3x2	3x3
#1	13,327	0.5	0.6	0.7	1.5	2.4	7.7	16,912	0.5	0.4	0.5	2.4	3.0	5.3
#2	38,932	4.1	1.3	1.9	4.0	3.6	5.2	49,172	4.1	1.2	1.2	7.5	4.8	5.5
#3	103,608	n/f	11.1	2.4	32.0	4.5	3.8	348,638	n/f	20.6	5.8	604.8	37.2	13.8
#4	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f	n/f
#5	25,024	n/f	4.3	2.9	23.8	7.9	9.4	36,738	1.4	1.4	1.2	40.7	24.4	27.8
#6	6,845	0.9	0.9	1.0	2.3	3.4	7.9	8,211	0.9	0.7	0.8	3.7	4.0	7.4
#7	14,196	n/f	3.1	1.8	9.6	7.0	22.4	18,263	n/f	12.5	17.1	147.9	482.0	357.9
#8	239,650	n/f	3.3	2.1	12.0	7.4	7.3	335,982	n/f	3.5	2.5	36.5	30.2	32.1
#9	104,174	n/f	4.5	3.1	n/f	14.0	27.0	6,558,767	n/f	3.5	1.3	n/f	4.6	2.2
#10	55,920	n/f	4.7	1.6	n/f	3.7	3.1	5,067,672	n/f	1.8	0.8	n/f	0.9	0.2
#11	4,958	n/f	6.1	10.8	n/f	35.6	28.7	7,516	n/f	8.0	16.5	n/f	224.8	191.0
#12	4,267,926	n/f	2.6	4.3	19.2	19.4	10.7	58,918,609	n/f	7.1	5.4	7.9	4.1	3.3
#13	2,348,863	n/f	3.3	5.0	n/f	44.7	24.2	n/f	n/f	n/f	n/f	n/f	n/f	11.6
#14	1,331,407	n/f	4.0	4.6	11.7	7.4	14.2	33,720,545	n/f	10.9	10.4	11.9	5.5	7.5
SLOW		**1.2**	**3.0**	**2.6**	**8.7**	**8.1**	**10.4**		**1.2**	**3.3**	**2.6**	**21.2**	**12.6**	**10.7**

Figure 6. Stateful exploration: mean number of transitions to the bug scenario. The last row shows the *geometric mean* of slowdowns.

Bug	Number of Paths							Number of Transitions						
	BLR (absolute)	Slowdown relative to BLR						BLR (absolute)	Slowdown relative to BLR					
		2x1	2x2	2x3	3x1	3x2	3x3		2x1	2x2	2x3	3x1	3x2	3x3
#1	17,683	0.5	0.5	0.5	4.1	4.7	4.4	712,757	0.4	0.7	1.0	5.5	10.8	13.9
#2	36,346	4.3	1.3	1.8	13.3	9.3	6.8	988,035	5.0	2.0	3.4	25.4	22.2	20.3
#3	144,314	n/f	12.4	2.0	83.0	7.5	4.8	8,407,268	n/f	21.4	3.6	159.0	16.3	13.2
#5	18,842	n/f	3.3	1.9	29.3	7.1	5.5	833,586	n/f	4.8	3.6	40.3	14.5	15.1
#6	7,887	0.6	0.6	0.5	3.9	4.5	4.8	315,681	0.3	0.5	0.6	3.6	6.9	10.4
#7	10,890	n/f	1.9	0.7	16.3	4.9	3.7	280,129	n/f	2.7	1.3	24.4	10.9	10.8
#8	232,071	n/f	3.8	2.4	25.3	16.2	15.2	9,067,584	n/f	5.8	4.6	41.5	37.2	45.6
#9	100,436	n/f	3.0	1.7	n/f	15.7	20.2	8,084,777	n/f	1.5	1.7	n/f	13.7	28.0
#10	34,950	n/f	4.8	1.4	n/f	6.2	3.1	3,624,957	n/f	1.9	1.1	n/f	4.4	3.3
#11	3,338	n/f	4.8	6.5	n/f	30.1	8.5	82,109	n/f	6.7	11.7	n/f	65.7	23.3
#13	2,072,099	n/f	0.9	1.0	n/f	7.7	3.6	376,786,069	n/f	0.6	1.1	n/f	7.9	5.5
#14	1,419,535	n/f	1.0	1.1	2.5	1.3	1.9	247,317,819	n/f	0.7	1.0	1.3	1.3	2.4
SLOW		**1.1**	**2.1**	**1.4**	**12.3**	**7.3**	**5.5**		**0.9**	**2.1**	**2.0**	**15.9**	**11.7**	**11.9**

Figure 7. Stateless exploration: mean exploration cost to the bug scenario. The last row shows the *geometric mean* of slowdowns.

B. Answering RQ2

We now look at how stable the BALLERINA speedups are over basic random for various types of exploration. So far we have considered CHESS-like, preemption-bounded exploration [3]. We consider three additional settings: exhaustive stateful search with partial-order reduction used by default in JPF [6], stateless search [21], and parallelized test execution where the tests are executed on several cores at once. We chose these settings as they are representative of what currently available tools like JPF or CHESS do. For parallelized test execution, we simulate what happens with four and eight cores, because these are the typical configurations for the current Intel i7 processors. For space reasons, we show only the *mean* values, not full box plots.

Figure 6 shows the results for CHESS-like and exhaustive JPF explorations. We have discussed the results for CHESS-like exploration. The results for exhaustive JPF exploration are similar. The absolute number of transitions required to find the bug is higher than for CHESS-like exploration, but BALLERINA still maintains its relative speedup over all basic random configurations. The average speedup is even somewhat larger (e.g., 3.3X vs. 3.0X for 2×2). Note that the Pool#13 bug is found by preemption-bounded exploration but not by exhaustive JPF exploration. This is due to the interaction of JPF's partial-order reduction (POR) and linearizability checking. Because JPF's POR could also affect the CHESS-like exploration, we turn off POR for this exploration; turning POR off for the exhaustive exploration would result in excessively high JPF runtime. We reported this behavior to JPF developers, and to the best of our knowledge, it did not affect any other experiment.

Figure 7 shows the results for stateless exploration based on re-execution. While JPF checkpoints and restores states to explore thread interleavings, stateless case considers an exploration that would re-execute the test to explore various thread interleavings, e.g., CHESS [3] or ReEx [35] tools do so. The left half of the table shows the exploration cost as the number of paths that the tool would need to execute for various tests to find the bug, while the right half shows the exploration cost as the number of transitions on these paths. These measures are commonly used to compare stateless techniques [35]. Again, exploring tests generated by BALLERINA is faster than exploring tests generated by basic random, but the average speedup is smaller than for stateful explorations. Note that stateless exploration by itself

Bug	BLR (rel. to 1-Core)		Bug	4-Core (Slowdown relative to BLR 4-Core)						8-Core (Slowdown relative to BLR 8-Core)					
	4-Core	8-Core		2x1	2x2	2x3	3x1	3x2	3x3	2x1	2x2	2x3	3x1	3x2	3x3
#1	3.9	8.7	#1	0.5	0.6	1.0	1.3	3.1	12.8	0.7	0.9	1.6	1.4	5.4	18.8
#2	3.7	7.6	#2	3.8	1.3	1.7	4.0	3.9	5.2	3.5	1.3	2.1	4.1	4.3	4.9
#3	4.1	8.3	#3	n/f	11.3	2.2	30.5	5.2	3.8	n/f	11.0	2.1	32.1	4.0	3.6
#5	4.0	8.2	#5	n/f	3.9	3.2	22.8	7.5	10.0	n/f	4.1	3.2	22.8	7.3	15.9
#6	4.0	7.6	#6	0.9	1.2	1.6	2.3	4.5	14.9	1.0	1.3	2.1	2.0	6.4	21.6
#7	4.5	9.7	#7	n/f	3.0	2.3	11.9	7.1	38.0	n/f	3.4	2.6	13.2	8.1	54.8
#8	4.0	7.8	#8	n/f	3.5	2.0	11.0	7.1	7.3	n/f	2.6	2.1	11.4	6.6	7.0
#9	4.6	9.5	#9	n/f	5.5	3.8	n/f	16.3	31.2	n/f	5.4	4.3	n/f	16.7	33.2
#10	8.3	26.5	#10	n/f	9.3	2.7	n/f	5.7	8.3	n/f	15.3	5.1	n/f	8.2	19.2
#11	4.1	7.0	#11	n/f	6.5	11.7	n/f	39.5	38.4	n/f	5.7	12.1	n/f	38.5	35.5
#12	5.1	10.3	#12	n/f	3.8	4.8	26.9	21.5	12.7	n/f	3.8	4.2	28.2	21.2	10.7
#13	3.7	7.9	#13	n/f	3.1	4.6	n/f	43.6	22.7	n/f	3.2	5.0	n/f	45.9	20.1
#14	4.8	10.8	#14	n/f	5.0	6.0	14.3	7.1	12.7	n/f	5.4	6.4	14.7	6.6	9.7
SPEED	**4.4**	**9.3**	**SLOW**	**1.2**	**3.4**	**3.0**	**9.1**	**9.0**	**13.2**	**1.3**	**3.6**	**3.4**	**9.3**	**9.8**	**15.0**

Figure 8. Parallelized test execution: mean exploration cost to the bug scenario.

Bug	BLR				2x2				2x3				3x2			
	fa/tb	#C	#IR (abs)	no-C (rel)	fa/tb	#C	#IR (abs)	no-C (rel)	fa/tb	#C	#IR (abs)	no-C (rel)	fa/tb	#C	#IR (abs)	no-C (rel)
#1	36	3	3.3	12.3	20	3	3.0	7.7	25	3	3.0	9.4	35	17	2.8	11.7
#2	67	2	1.3	44.9	25	4	1.2	16.1	28	13	4.6	6.6	52	28	3.5	13.4
#3	0	1	1.0	1.0	0	1	1.0	1.0	144	10	1.8	61.5	175	29	4.2	39.7
#5	0	1	1.0	1.0	0	2	1.0	1.0	0	2	1.0	1.0	0	6	1.0	1.0
#6	0	2	1.0	1.0	0.17	7	1.4	0.8	1	9	1.6	1.0	1	19	2.5	0.7
#7	0	1	1.0	1.0	0	2	1.0	1.0	0	2	1.0	1.0	0	6	1.0	1.0
#8	50	4	1.3	37.3	77	8	3.2	21.7	94	31	9.3	8.4	341	73	15.6	23.6
#9	116	5	4.9	14.8	85	3	2.7	17.8	80	6	7.6	7.0	538	17	11.6	45.6
#10	0	2	1.0	1.0	0	2	1.0	1.0	0	4	1.0	1.0	0	9	1.0	1.0
#11	0	1	1.0	1.0	0	1	1.0	1.0	6	2	1.9	2.5	0	2	1.0	1.0
#12	0	1	1.0	1.0	289	7	5.4	47.9	630	15	14.8	48.8	935	50	20.7	51.8
#13	113	3	2.6	34.3	318	7	7.0	19.5	894	14	27.3	24.7	5,830	39	16.2	244.0
#14	47	3	3.1	15.0	122	8	5.5	15.1	241	12	26.4	7.3	288	38	16.1	16.4
SLOW				**4.0**				**4.7**				**5.4**				**8.2**

Figure 9. Ratio of false alarms and true bugs, # of clusters, # of reports inspected with clustering, improvement over no clustering.

is slower than stateful for our experiments, and Pool#12 does not even finish in a reasonable time for stateless.

Figure 8 shows the results for exploring the generated tests in parallel, on four and eight cores. Note that we do *not parallelize exploration of one test at a time* but rather *explore in parallel several tests at once*. Each test is explored with a CHESS-like preemption bounding as in Figure 5. The first part of the table shows the speedup that this parallelized test execution achieves over execution on one core for tests generated by BALLERINA. The average speedup is super-linear (4.4X on 4 cores and 9.3X on eight cores). This is not surprising for search problems [36], because the search finishes as soon as one core finds the bug. The second part of the table shows the slowdown of the tests generated by basic random compared to the tests generated by BALLERINA, when run on *the same number of cores*. Comparing the average slowdown to the case for one core (Figure 6), we find that BALLERINA performs even better than basic random when test execution is parallelized (e.g., for 2×3 the slowdown is 2.6X on one core, 3.0X on four cores, and 3.4X on eight cores). This is important as the availability of multi-core processors means that test executions in the future are likely to be parallelized.

Tests generated by BALLERINA *find bugs faster than tests generated by basic random configurations for a variety of different exploration approaches.*

C. Answering RQ3

Since reports of linearizability violations can be false alarms or true bugs, developers need to inspect a number of reports before finding a true bug. Figure 9 shows how our clustering helps with reducing the number of inspections. For each bug and several test generation configurations (the results are similar for the configurations not shown), we tabulate the ratio of false alarms to true bugs before clustering ('fa/tb'), the number of clusters that our technique computes, the expected number of inspections with clustering (computed as the arithmetic mean of 50 random orderings of reports), and the ratio of the expected number of inspections *without* clustering over the expected number of inspections *with* clustering. Effectively, the latter ratio shows the benefit that our clustering provides over the base case with no clustering, and the last row shows the *geometric mean* average of this benefit.

The improvement ranges from 1.0 (when there are no false alarms and hence the number of inspections is exactly one both with and without clustering) up to 244.0X (for Pool#13 and 3×2). Note that the number of inspections with clustering is never worse than the number of inspections without clustering for tests generated by BALLERINA.

Our clustering technique reduces the number of inspected reports for tests generated by both BALLERINA *and basic random generation, on average 4X-8X.*

D. Threats to Validity

Internal threats: We conducted our experiments with the default settings of JPF (version 6.0). As explained in Section V-B, we encountered an incompatibility of JPF's POR and linearizability checking for Pool#13, but to the best of our knowledge, it did not affect any other experiment. However, changing other JPF settings could affect the results that we obtained.

External threats: The code under test and bugs that we use are from a variety of sources and diverse in terms of the statistics shown in Figure 4. However, we cannot guarantee that they form a representative sample of bugs in multithreaded Java code. To mitigate the limitation of using one particular exploration, we evaluated BALLERINA with four different explorations.

Construct threats: We measure exploration cost with the number of transitions (and paths for re-execution) instead of real time. This is common in previous related studies [17], [34]. We measure inspection effort with the number of reports which is a proxy for human time. This is common in previous related studies [24], [27].

Conclusion threats: The number of random seeds (200 for generation and 50 for clustering) that we used may not be sufficient to accurately characterize real distribution of these random processes.

VI. RELATED WORK

There is a rich body of work on random test generation for *sequential code* [14], [37], [38], including combinations with static analysis [15], [39], symbolic execution [40], and search-based techniques [13], [41], [42]. Our BALLERINA technique utilizes Randoop [14] and modifies it to more densely cover states of objects for the given CUT. Related to this, techniques based on adaptive random testing [43]–[45] use various measures for object distance to generate more divergent test inputs. However, unlike those projects, BALLERINA generates tests for *multithreaded code*.

For finding bug-triggering *interleavings* in multithreaded code, numerous techniques have been proposed, including static [46] and dynamic approaches [47], and their combination [33], [48]. Randomized thread scheduling and statistical fault localization have also shown promise in testing parallel code [17]–[20]. However, all those techniques assume that the test code is provided and only *explore provided tests*. In contrast, BALLERINA automatically *generates multithreaded tests* that can expose bugs. BALLERINA also employs search for linearizability violations, inspired by Line-Up [2].

Environment generation for multithreaded code is related to generation of test code. For example, Tkachuk and Rajan [49], [50] automatically generate driver and stub for the system under test based on formal specification of the system properties. BALLERINA does not require the user to explicitly provide formal specifications but uses random testing to generate driver code and uses linearizability as an implicit specification for the CUT. BALLERINA also clusters failures to reduce the number of inspections of false alarms.

Researchers have proposed different techniques for clustering failing runs. Most previous work clusters failures based on *execution profiles* from monitored runs. Podgurski et al. use feature selection to train clusters based on execution profiles, which are used to group similar failures [24], [27]. Jones et al. combine fault-localization information with profiles to cluster failing tests [25]. Yoo et al. incorporate human expert knowledge into clustering [28]. Zheng et al. use statistical methods to find super-bug predictors in multiple faults setting [29]. Different from the previous work, our clustering technique focuses on grouping failures of *multithreaded tests* based on the methods executing in parallel and the type of failure.

VII. CONCLUSIONS

Testing multithreaded code is becoming more important but remains challenging and costly. Automated testing can help to reduce the costs, but most research focuses on automated exploration of thread interleavings for manually written test code. We have presented BALLERINA, a novel technique that automates generation of multithreaded unit test code. We have also presented a technique for clustering failures of automatically generated multithreaded tests. The experiments with 14 bugs show that BALLERINA can trigger bugs substantially faster than basic random generation, and that our clustering can greatly reduce the number of test failures that need to be inspected to find a true bug. Our experiments exposed three previously unknown bugs, two of which were already fixed. While random generation showed promising results, it would be useful to consider a more guided search for test generation.

ACKNOWLEDGMENTS

We thank Vilas Jagannath for extensive discussions about this work. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-1012759, CNS-0958199, CCF-0916893, and CCF-0746856, by Intel under the Illinois-Intel Parallelism Center (I2PC), and by the Swiss National Science Foundation under grant number 200021-134453.

REFERENCES

[1] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer, "Preemption sealing for efficient concurrency testing," in *TACAS*, 2010.

[2] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-Up: A complete and automatic linearizability checker," in *PLDI*, 2010.

[3] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *PLDI*, 2007.

[4] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *ASPLOS*, 2009.

[5] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic replay with execution sketching on multiprocessors," in *SOSP*, 2009.

[6] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," *ASEJ*, 2003.

[7] J. Burnim, T. Elmas, G. C. Necula, and K. Sen, "NDSeq: Runtime checking for nondeterministic sequential specifications of parallel correctness," in *PLDI*, 2011.

[8] J. Burnim, G. C. Necula, and K. Sen, "Specifying and checking semantic atomicity for multithreaded programs," in *ASPLOS*, 2011.

[9] J. Burnim and K. Sen, "Asserting and checking determinism for multithreaded programs," in *ESEC/FSE*, 2009.

[10] ——, "DETERMIN: Inferring likely deterministic specifications of multithreaded programs," in *ICSE*, 2010.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006.

[12] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," in *PLDI*, 2009.

[13] L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: An evolutionary test approach for Java," in *ICST*, 2010.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.

[15] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *ISSTA*, 2011.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[17] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *ICSE*, 2007.

[18] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," *IBM Systems Journal*, 2002.

[19] K. Sen, "Race directed random testing of concurrent programs," in *PLDI*, 2008.

[20] ——, "Effective random testing of concurrent programs," in *ASE*, 2007.

[21] M. Musuvathi and S. Qadeer, "Fair stateless model checking," in *PLDI*, 2008.

[22] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, 1990.

[23] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *PLDI*, 2007.

[24] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *ICSE*, 2001.

[25] J. A. Jones, M. J. Harrold, and J. F. Bowring, "Debugging in parallel," in *ISSTA*, 2007.

[26] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE TSE*, 2008.

[27] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *ICSE*, 2003.

[28] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *ISSTA*, 2009.

[29] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *ICML*, 2006.

[30] Apache Software Foundation, "POOL-184," https://issues.apache.org/jira/browse/POOL-184.

[31] ——, "POOL-189," https://issues.apache.org/jira/browse/POOL-189.

[32] ——, "LOG4J-51783," https://issues.apache.org/bugzilla/show_bug.cgi?id=51783.

[33] F. Chen, T. F. Şerbănuţă, and G. Roşu, "jPredictor: A predictive runtime analysis tool for Java," in *ICSE*, 2008.

[34] M. B. Dwyer, S. Person, and S. G. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *FSE*, 2006.

[35] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *ISSTA*, 2011.

[36] V. N. Rao and V. Kumar, "Superlinear speedup in parallel state-space search," in *FOSTTCS*, 1988.

[37] C. Csallner and Y. Smaragdakis, "DSD-Crasher: A hybrid analysis tool for bug finding," in *ISSTA*, 2006.

[38] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object capture-based automated testing," in *ISSTA*, 2010.

[39] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *OOPSLA*, 2011.

[40] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.

[41] J. H. Andrews, T. Menzies, and F. C. H. Li, "Genetic algorithms for randomized unit testing," *IEEE TSE*, 2011.

[42] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *FSE*, 2010.

[43] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *JSS*, 2010.

[44] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *ICSE*, 2008.

[45] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of Java programs," in *ASE*, 2009.

[46] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI*, 2006.

[47] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *PLDI*, 2009.

[48] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP*, 2003.

[49] O. Tkachuk and S. P. Rajan, "Application of automated environment generation to commercial software," in *ISSTA*, 2006.

[50] ——, "Combining environment generation and slicing for modular software model checking," in *ASE*, 2007.