EVALUATING MACHINE-INDEPENDENT METRICS FOR STATE-SPACE EXPLORATION

BY

MATHEW ALAN KIRN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Associate Professor Darko Marinov

Abstract

Many recent advancements in testing concurrent programs have surfaced as novel optimization and heuristic techniques in tools that explore the state spaces of tests for such programs. To empirically evaluate these techniques, researchers apply them on subject programs, capture a set of metrics, and compare these metrics to provide some measure of the techniques' effectiveness. From a user's perspective, the metric that best measures effectiveness is the amount of real time to find a bug (if one exists), but using real time for comparison can produce misleading results because it is necessarily dependent on the configuration of the machine used (i.e., hardware, OS, etc.). The metrics used in evaluations in the literature often vary widely and are either machine-independent (e.g., number of states, transitions, paths) or machine-dependent (e.g., real time, memory), and are captured using a variety of machine configurations ranging from a single machine to a cluster of machines. Depending upon the machine configuration(s) and metric(s) selected for a particular evaluation, the results may suggest different conclusions, and the experiments may be difficult to repeat. As a result, it can be difficult to perform meaningful comparisons for state-space exploration tools and the techniques they employ.

This thesis provides a study of the usefulness of different metrics and machine configurations for two different state-space exploration frameworks for Java, JPF (stateful) and ReEx (stateless), by revisiting and extending a previous study (Parallel Randomized State-Space Search) and evaluating the correlation of several machine-independent metrics with real time. We have conducted a set of experiments across both previously used and new subject programs in order to evaluate the degree to which several machine-independent met-

rics correlate with real time both on a single machine and on a high-performance cluster of machines. We provide new evidence for selecting metrics in future evaluations of state-space exploration techniques by showing that several machine-independent metrics for state-space exploration are a good substitute for real time, and that reporting real time results even from clusters of machines can provide useful information.

To my family.

Acknowledgments

I would like to thank:

- My parents and my brother for their love and support.
- Prof. Darko Marinov for being a dedicated mentor and for sharing his knowledge, advice, and support.
- Vilas Jagannath for many insightful conversations and ultimately a collaboration with both myself and Yu Lin that resulted in this thesis.
- Many great professors under whom I have had the privilege to learn and grow as a student, including Prof. Darko Marinov, Prof. Madhu Parthasarathy, Prof. Sam King, Prof. Vikram Adve, Prof. Marco Caccamo, and others.
- Members of my research group and other colleagues with whom I have had a chance to interact and who have exposed me to other interesting research and domains of study, including Vilas Jagannath, Milos Gligoric, Qingzhou Luo, Adrian Nistor, Shin Hwei Tan, Steven Lauterburg, Brett Daniel, Rajesh Karmani, Samira Tasharofi, and others.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit
DFS	Depth-First Search
GB	Gigabytes
GHz	Gigahertz
ICB	Iterative Context-Bounding
JPF	Java PathFinder
JVM	Java Virtual Machine
OS	Operating System
PDR	Point of Diminishing Returns
PRSS	Parallel Randomized State-Space Search
RAM	Random Access Memory
ReEx	Re-Execution based Explorer
RQ	Research Question

Chapter 1

Introduction

It is increasingly important to test concurrent programs as these programs are being developed and used more widely with the prevalence of multicore processors. Concurrent programs are difficult to develop as they are notorious for having hard-to-find and hard-to-reproduce bugs. It is challenging for developers to avoid such bugs due to the complexity of reasoning about an enormous set of possible interleavings that are not immediately intuitive from the code itself. Concurrent programs are also difficult to test as it is necessary to check how a program behaves not only for different inputs but also for different interleavings for a given input. As such, it is important to develop and evaluate efficient techniques that test concurrent code effectively.

Concurrent programs are often tested using tools that systematically explore possible interleavings of a given program for a given input—this approach is known as *state-space exploration* [7]. Conceptually, the exploration begins from a start state, executes the program up to a point where a set of non-deterministic choices $c_1, \ldots, c_n$ are possible (e.g., two or more threads are enabled so either one of them could proceed first), selects one of the choices c_i, continues exploration based upon that choice until a particular criteria is satisfied, and then backtracks to c_i in order to explore the choices extending from c_{i+1}. This approach can result in a huge number of cases to explore (e.g., if there are n enabled threads, there could be up to $n!$ different results of execution based on the orderings of these threads)—this problem is known as state-space explosion.

Since state-space exploration is essentially a search over the state-space, tools can adopt different *search strategies* (e.g., depth-first, random, or best-first) to perform the exploration.

Also, tools can employ different techniques to restore states in which other choices were available, in order to explore those remaining choices. Some tools *checkpoint* encountered states and restore states by restoring their checkpoints. Other tools store the choices that lead to encountered states and restore states by *re-executing* those choices. While checkpointing states can be memory and time intensive, hashes of state checkpoints can be used to remember previously explored states. Tools that remember previously explored states are called *stateful* tools, and they can reduce exploration time by avoiding re-exploration of states. Tools that do not remember previously explored states are called *stateless* tools, and they can reduce exploration time by avoiding costly comparisons for encountered states.

Developing new techniques for faster state-space exploration has received much attention in research, e.g., [8, 10, 11, 13, 14, 16, 21]. Researchers have recently proposed many novel optimization and heuristic techniques to reduce the costs of exploring large state spaces, and have implemented these techniques in several tools. These techniques can be categorized in many different ways. One category of techniques consists of strategies that prioritize or select the exploration of certain parts of the state space by leveraging additional information or heuristics [8, 12, 14, 16, 18, 21, 23, 26]. A recent, promising strategy in this category is known as *Iterative Context-Bounding (ICB)* [16], which prioritizes exploration according to the number of context switches between threads. The main idea is to explore the parts of the state space that consist of a small number of context switches, because many concurrency bugs often manifest in such schedules. We use an implementation of the ICB strategy in the ReEx framework [14, 20] for some of our experiments. Other techniques include those that parallelize a single exploration by partitioning into non-overlapping subspaces [5, 6] and those that exploit diversity among multiple different (potentially overlapping) complete explorations by performing them in parallel [10, 13].

To empirically evaluate these techniques, researchers apply them on subject programs, capture a set of metrics, and compare these metrics to provide some measure of the techniques' effectiveness. A measure of effectiveness usually has two dimensions. One dimension

is whether a technique provides a high level of assurance with respect to a particular criteria. For example, given a fixed input to a concurrent program, a user may want complete assurance that no deadlocks can occur in all schedules within any two context switches. The second dimension is whether a technique finishes quickly. A user will have to allocate precious resources (e.g., wait a long time or spend more money on faster machines) that will be consumed in order to perform the state-space exploration, so a technique that provides high assurance in a short amount of real time will consume less of a user's resources. However, real time is only a useful measurement for a given machine configuration (e.g., how fast is my CPU/memory/disk/network, how are tasks scheduled, when will garbage collection/disk paging occur, etc.). As such, research studies that provide evaluations in real time are necessarily machine-dependent, and thus their results may not apply across different machine configurations, may be hard to repeat, and may be difficult to compare with other techniques.

Studies that perform large experiments for comparing techniques on clusters of machines are further impacted by this concern since the results from clusters may not necessarily allow comparisons of real time. This leads researchers to use machine-independent metrics that allow comparison of techniques across different machine configurations, but empirical studies in the literature often vary widely in their selection of *machine-independent metrics* (e.g., number of states, transitions, or paths explored) and *machine-dependent metrics* (e.g., real time or amount of memory taken for exploration). When machine-independent metrics are used, they do not provide much utility in terms of evaluating the efficiency of techniques if they are not good indicators of real time. Hence, machine-independent metrics that correlate highly with real time are most desirable.

Dwyer et al. performed an important study on controlling factors in evaluating state-space exploration techniques [11]. The study found that three factors—the default search strategy used, the error density of the state space, and the number of threads—greatly affect comparisons among techniques. Based on this study, Dwyer et al. selected a set of

programs to be used for comparing techniques. Moreover, based on the study, they proposed and evaluated a new technique called Parallel Randomized State-Space Search (PRSS) [10]. PRSS was evaluated for the Java PathFinder (JPF) tool for stateful exploration of Java programs [15, 25] (specifically JPF version 3.1.2).

In this thesis, (i) we revisit and extend the PRSS study, and (ii) we additionally evaluate the correlation of machine-independent metrics with real time for two different state-space exploration tools for Java, stateful JPF (the latest version 6.0) and stateless ReEx (version 1.0). We have conducted a set of experiments across both previously used and new subject programs in order to evaluate the degree to which several machine-independent metrics correlate with real time both on a single dedicated desktop and on a high-performance compute cluster.

In summary, our results provide new evidence for selecting metrics for future evaluations of state-space exploration techniques. We conducted our experiments using 13 concurrent Java programs that exhibit several different types of errors. We have several findings. First, for all programs evaluated with JPF, each metric has greater than 50% correlation with time on a desktop machine, where all but 2 programs exhibit greater than 86% correlation. Second, for most programs evaluated with JPF, each metric has greater than 50% correlation with time on a cluster. Third, for all but one program evaluated with ReEx, each metric has greater than 79% correlation with time on a desktop machine. Fourth, for all programs evaluated with ReEx, each metric has greater than 50% correlation with time on a cluster, where all but one program exhibit greater than 87% correlation. Fifth, all of the currently widely used machine-independent metrics for state-space exploration are equally good proxies for real time, so researchers can measure and report metrics that have the lowest measurement overhead like states for stateful exploration and schedules for stateless exploration. Sixth, real time measurements from clusters do correlate reasonably well with machine-independent metrics, which is surprising, and so should be reported by studies that perform experiments on clusters.

The rest of this thesis is organized as follows. Chapter 2 provides background information about the state-space exploration tools used in our study, reviews related work on using machine-dependent and machine-independent metrics for evaluating state-space exploration, and provides more information on the PRSS study. Chapter 3 presents the details of the study, and we present the results in Section 3.4. Chapter 4 concludes the thesis.

Chapter 2

Background

This chapter presents background information that will provide context for discussions later in this thesis. We begin by describing the operation of a recent stateless exploration tool, ReEx, with a simple example. Then, we provide an overview of a stateful exploration tool, JPF, and compare and contrast its capabilities to ReEx. Furthermore, for both ReEx and JPF, we define their commonly used metrics and describe the ways in which researchers use these metrics in comparing techniques. Finally, we provide an overview of the PRSS study that motivates our work.

2.1 ReEx

ReEx [14, 20] is a general-purpose state-space exploration tool that supports exploration of a large class of multi-threaded Java programs. It takes as input a multi-threaded Java program and optional attributes, such as the scheduling strategy to use, and produces as output whether or not an error was detected during exploration. ReEx also outputs a set of metrics that summarize the performed exploration; these metrics are described in Section 2.3. ReEx can check implicit safety properties defined by the JVM, such as null pointer exceptions, as well as user defined properties in the form of explicit assertions. ReEx supports the exploration of both thread choices (which come from non-determinism in scheduling threads) and data choices (which come from explicit non-determinism that programmers can add to the code), but we will focus our discussion on thread choices. ReEx also supports continuing exploration even after errors have been detected, which is useful,

for example, for gathering information about the state space of the program under test.

ReEx itself is just a Java program that executes on a standard JVM and internally controls the exploration of a specified Java program by performing dynamic bytecode instrumentation using the ASM library [4]. The instrumentation enables exploration by intercepting schedule-relevant events in the program being explored. A *schedule-relevant event* is the execution of a bytecode/instruction that has side-effects that can affect the behavior of other enabled threads. For example, in a program with two or more threads enabled, the acquisition of a lock by one of the threads is a schedule-relevant event; different results can be obtained if another thread is scheduled before the lock acquisition versus after the lock acquisition. In order to perform the exploration, ReEx rewrites the bytecode for a class (dynamically at load time), such that an internal scheduler is informed about all schedule-relevant events when they occur. This allows the internal scheduler to collect necessary information about all the enabled threads and orchestrate exploration. The internal scheduler runs in a loop where it first allows all threads to reach schedule-relevant events and at that juncture (called a *choice* when two or more threads are enabled) asks a *scheduling strategy* to decide which of the threads should be allowed to proceed. ReEx makes choices for a single schedule until the end of the program is reached, at which point ReEx restarts exploring from the beginning of the program by re-executing it. ReEx continues this process until either (i) an error is detected, (ii) the scheduling strategy decides that there are no more choices left to explore, or (iii) a timeout occurs.

An example that helps illustrate the operation of ReEx is shown through Figures 2.1 and 2.2. Figure 2.1 shows a simple Java program in which a deadlock error can manifest. Even this simple program has 150 possible schedules due to accesses to shared fields `a` and `b`, lock acquisitions, and thread ordering. Figure 2.2 shows a partial exploration of this program for one full schedule that leads to the deadlock error under the default ICB scheduling strategy. In Figure 2.2, nodes represent schedule-relevant *events*, labeled edges represent a *choice* that ReEx can make between two or more events, and non-labeled edges represent a transition

```
1  public class DeadlockSimple {
2      private static Object a = new Object();
3      private static Object b = new Object();
4      public static void main(String[] args) {
5          new Thread() {
6              @Override
7              public void run() {
8                  synchronized (a) {
9                      synchronized (b) {
10                     }
11                 }
12             }
13         }.start();
14         synchronized (b) {
15             synchronized (a) {
16             }
17         }
18     }
19 }
```

Figure 2.1: Example Java code illustrating a simple deadlock error

to a new event for which there are no other choices. Each node is labeled "[⟨Event Type⟩] ⟨Object⟩(⟨Line Number⟩)", where ⟨Event Type⟩ can be FA to denote a field access or LK to denote a lock acquire operation, ⟨Object⟩ is the corresponding field from Figure 2.1 upon which the ⟨Event Type⟩ will be performed, and ⟨Line Number⟩ is the line number of the code from Figure 2.1 where the event occurs. ReEx explores this schedule by starting from an initial state and making choices that result in a lock dependence cycle among the threads that are blocked waiting for locks, i.e., resulting in a deadlock. Note that the last two events in the schedule must be executed and that there are no alternative events to choose from; this is an important distinction between the total number of choices and the total number of events in an exploration as we will describe further in in Section 2.3.

While ReEx is useful for finding errors like these in many multi-threaded Java programs, it does have some limitations. Since ReEx is stateless, it is not capable of determining whether a state has already been explored and therefore may explore a state many times, adding to the amount of time taken to perform an exploration. For programs with cyclic state spaces, i.e., programs that can execute indefinitely, this limitation can restrict ReEx to only narrowly exploring one single schedule for a long time, thus preventing ReEx from exploring other schedules which may contain errors sooner. This limitation could partly be

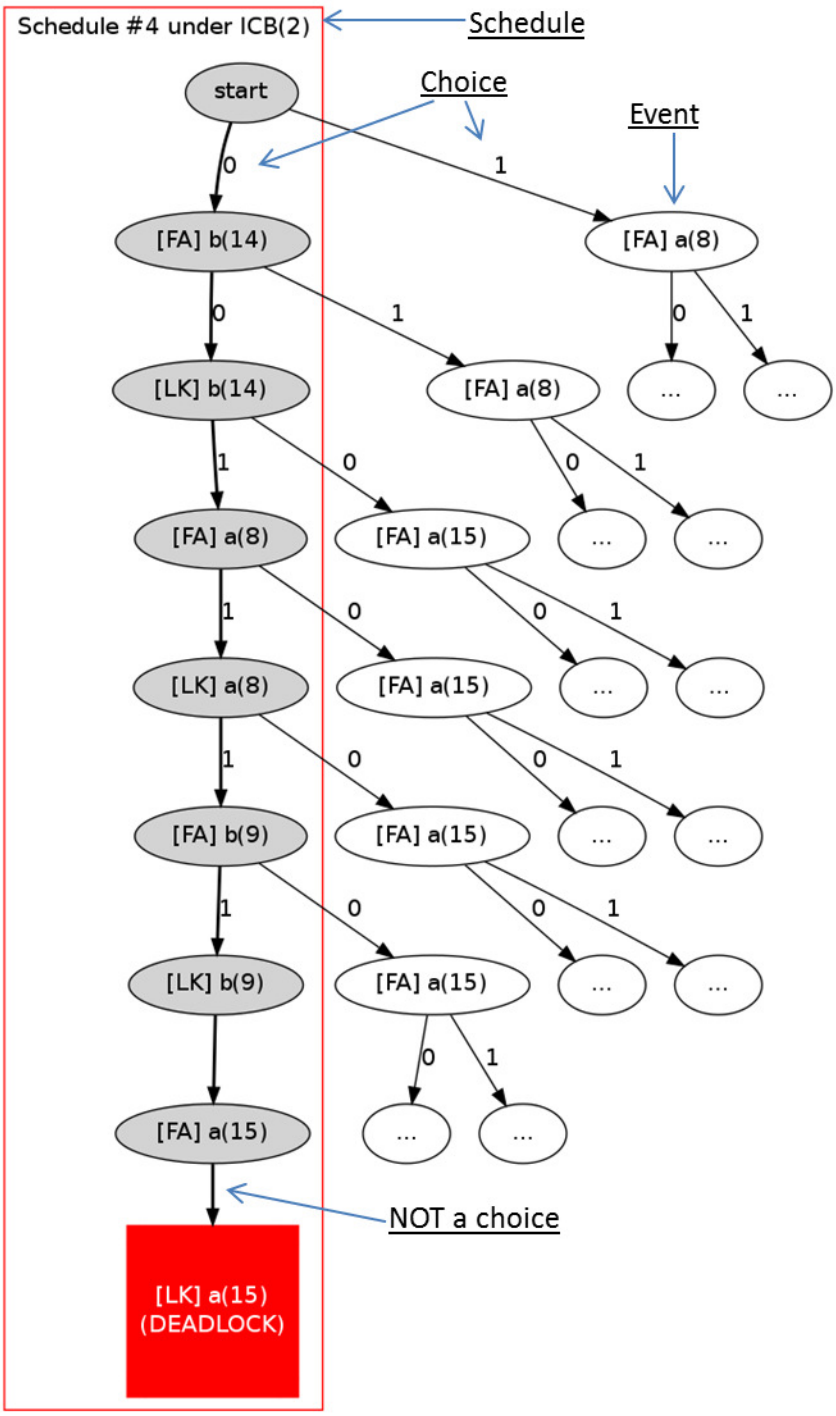

Figure 2.2: Graphical representation of a partial exploration performed by ReEx

addressed using timeout mechanisms to temporarily stop an exploration, serialize the list of explored choices, and then resuming exploration by deserializing the list of choices and continuing an exploration along a different path. However, ReEx does not currently support

resuming an exploration from a previously serialized list of choices. Another way to partly address this limitation would be to enforce fair scheduling [17], but ReEx does not currently support fair scheduling. Alternatively, a practical approach to addressing this limitation is to run multiple ReEx instances in parallel with randomized scheduling strategies, and our study suggests that this is an effective strategy.

2.2 Java PathFinder

In this section, we provide a brief overview of Java PathFinder (JPF) and focus more on how it compares with ReEx. JPF is covered in more detail elsewhere [15, 25].

JPF is a stateful state-space exploration tool that supports the exploration of Java programs for which native methods are modeled. The inputs and outputs to JPF, as well as the types of errors that can be detected, are the same as described for ReEx in Section 2.1. More specifically, JPF is a JVM implemented in Java that interprets bytecodes and maintains system state much the same way as a standard JVM, but exposes mechanisms such as scheduling strategies for controlling an exploration. In contrast to instrumenting Java bytecode that runs on a standard JVM, as performed by ReEx, JPF works by interpreting each bytecode instruction itself. Furthermore, JPF handles native method invocations by defining a set of interfaces that model expected behavior, but these interfaces must be defined for every native method that can be executed by a program, which can be difficult and time consuming to implement correctly.

JPF differs from ReEx in that JPF is itself a JVM and therefore must implement the required functionality of a JVM including interpreting every bytecode instruction and handling native methods, which involves additional overhead. However, this allows JPF to maintain explicit program data information, such as static area and heap, which it uses to store, restore, and compare states in order to avoid re-exploring the same state multiple times. Maintaining this additional data, though, also incurs overhead, even despite JPF's usage of

special data structures that facilitate performing these operations efficiently. In contrast, ReEx just executes on a standard JVM so there is no need for it to perform expensive state maintenance operations, and ReEx can handle native methods without any additional support. The only information that ReEx maintains is the execution states of all live threads (e.g., blocked or running) along with information about the choices that have been made during an exploration (which is maintained by the scheduling strategy). However, this precludes ReEx from being able to prevent the re-exploration of JVM states that have been previously explored and that may appear in multiple schedules.

JPF also differs from ReEx in the types of state-space metrics that it reports. Since JPF is stateful, its machine-independent metrics include program states and transitions between them. Also, since JPF is a JVM, it can report fine-grained metrics such as the number of bytecode instructions executed, without incurring any significant overhead. JPF can also report machine-dependent metrics such as time taken for an entire exploration. Metrics for JPF and ReEx are described more in Section 2.3.

2.3 Metrics

This section highlights the importance of selecting metrics to report in research evaluations. We also define the metrics that are commonly used to summarize explorations for stateful and stateless state-space exploration tools.

Researchers developing techniques for improving state-space exploration use various metrics to evaluate their techniques and present their results. Table 2.1 shows a sampling of recent papers presenting techniques for state-space exploration and the metrics used in their evaluation.

Since researchers recognize that end users care most about the time it takes to find a bug, some researchers choose to report time as a way to evaluate techniques. Also, since memory can sometimes be traded off for time, some evaluations may choose to report the memory

consumed during exploration to provide more perspective on the performance characteristics of a technique. Both time and memory are machine-dependent metrics, i.e., their values can change depending upon the machine used for exploration. Thus, their values must be considered in the context of the machine on which they are collected. As noted from Table 2.1, some evaluations do not even mention any details of the machine used when reporting machine-dependent metrics. As such, their results may be interpreted incorrectly, and future research cannot meaningfully build upon those evaluations. This demonstrates the importance of selecting metrics properly and motivates the need for machine-independent metrics in particular.

Researchers also use machine-independent metrics to evaluate their techniques. This allows for analysis of results across machine configurations. Furthermore, some researchers perform experiments on clusters in order to increase the number of experiments conducted or to decrease the amount of time necessary to complete the experiments. These clusters are usually composed of a heterogeneous mixture of machine configurations managed by a centralized load balancing system. Thus, it may be hard to compare machine-dependent metrics from these types of experiments, so machine-independent metrics are usually chosen, though not always. Machine-independent metrics vary based on the type and implementation of the tool used for exploration. In this thesis, we consider both stateful (JPF) and stateless (ReEx) exploration tools.

Common machine-independent metrics for stateful exploration (e.g., JPF) include:

- **States**: The total number of unique program states encountered during the exploration.
- **Transitions**: The total number of transitions performed during the exploration. A transition progresses the execution from one program state to another.
- **Instructions**: The total number of instructions executed during the exploration. Each transition consists of one or more instructions.

Paper	Metric	Search Type	Used Randomness	Machine Configuration
CAPP [14]	Transitions	Stateful	Yes	Cluster
CAPP [14]	Schedules	Stateless	Yes	Cluster
Controlling Factors [11]	States	Stateful	Yes	Cluster
CTrigger [19]	Time	Stateless	No	Desktop machine
Depth Bounding [23]	States, Transitions, Time, Memory	Stateful	No	Not specified
Distributed Reachability [5]	Time	Stateless	Yes	Cluster
Gambit [8]	Time, Memory	Hybrid	Yes	Not specified
ICB [16]	States, Time	Hybrid	No	Not specified
PENELOPE [22]	Time	Stateless	No	Not specified
PRSS [10]	States	Stateful	Yes	Cluster
Random Backtracking [18]	States, Transitions	Stateful	Yes	Not specified
Swarm [13]	States, Time, Memory	Stateful	Yes	Desktop Machine

Table 2.1: Attributes of evaluation for some recent papers on state-space exploration

- **ThreadCGs**: The total number of thread choices generated during exploration. A thread choice is generated when the exploration encounters a state with two or more enabled threads.

Common machine-independent metrics for stateless exploration (e.g., ReEx) include:

- **Schedules**: The total number of schedules encountered during the exploration. Since the most common form of stateless exploration performs backtracking via re-execution, each schedule involves the re-execution of the program being explored for a unique sequence of choices.

- **Choices**: The total number of choices encountered during the exploration. Each choice is a point during an execution where more than one thread is available to be scheduled, and one of them is chosen to be scheduled. As previously noted in Section 2.1, if only one thread is available to be scheduled, then this is not considered a choice.

- **Events**: The total number of events encountered during the exploration. Each event denotes the execution of a scheduling relevant bytecode instruction (e.g., lock/unlock, field read/write). Choices are created when two or more threads in the program are all about to perform an event.

- **Threads**: The total number of threads created during the exploration across all schedules. The threads that are part of the program are re-created during each schedule.

In Chapter 3, we investigate the correlation of these commonly used machine-independent metrics with real time, which is the metric that a user of a tool eventually experiences and cares most about.

2.4 Parallel Randomized State-Space Search

The Parallel Randomized State-Space Search (PRSS) technique consists of performing multiple parallel randomized state-space explorations, each with a unique random seed, across different machines on a cluster and stopping all the explorations when one of them detects a failure. The intuition behind the technique is that different randomized explorations will exercise diverse regions of the state space and hence detect a failure (if one exists) faster than just performing a single non-randomized (default) exploration.

PRSS was introduced by Dwyer et al. [10] and was originally implemented and evaluated with randomized depth-first explorations performed using JPF (specifically, an older version, JPF 3.1.2). It was shown that PRSS could achieve significant speedups (more than 100x) in exploration time for various subject programs using a reasonably small number (5-20) of parallel machines. The rationale behind the effectiveness of PRSS is that the distribution of randomized explorations can contain many explorations that detect a failure faster than the default non-randomized exploration. Hence, performing multiple randomized explorations in parallel increases the probability of one of the explorations being faster than the default exploration and thus achieving a speedup in exploration time.

To evaluate PRSS, the authors performed the following [10]:

- Ran 5000 randomized depth-first explorations of each artifact using JPF.
- Sampled 50 random simulations of various PRSS configurations including 1, 2, 5, 10,

15, 20, and 25 parallel computers for each artifact. For example, the simulations for the PRSS configuration with 2 parallel computers involved randomly choosing 50 pairs of explorations from logs of the 5000 explorations recorded for an artifact and retaining the fastest exploration within each pair.

- Plotted the distribution of the results of the 50 random simulations for each PRSS configuration and each artifact. The mean of the distribution was considered the expected performance.

- Determined (through a consensus among the authors) the point of diminishing return (PDR), i.e., the configuration where adding more parallel computers did not yield substantial performance benefits compared to the cost of utilizing additional computers.

Through this evaluation, PRSS was shown to be effective for exploring various concurrent programs using stateful depth-first exploration performed with JPF version 3.1.2. However, it is not clear whether similar results could be obtained for (i) other concurrent programs, (ii) stateless exploration, (iii) different search strategy, or (iv) the latest version of JPF which incorporates many new optimizations. We revisit PRSS with these four additional contexts in Chapter 3.

The authors of the PRSS study also considered the use of stateless search using JPF, but they decided not to evaluate stateless search because the version of JPF that they used could not find an error in orders of magnitude more time than stateful searches that could find the error in a few minutes. In our evaluation, we perform stateless search with ReEx and find that stateless search can be used effectively for PRSS, and that its effectiveness may depend on the implementation of the tool used.

Chapter 3

Study

3.1 Study Goals

The goal of our study is two-fold. First, we reinvestigate PRSS with the new contexts: different version of JPF (the latest version, 6.0), stateless exploration (using ReEx), different search strategy (ICB), and different concurrent programs. Second, we utilize the results of these and additional experiments to answer questions about the correlation of various machine-independent metrics with real time both on a compute cluster and on a dedicated desktop computer. As such, we revisit the same three research questions from PRSS [10] and address two more for correlation of real time with machine-independent metrics. Namely, we address the following five questions:

RQ1 - Cost Reduction: Does there exist a feasible PRSS configuration that performs better than the default exploration? A feasible configuration is one with a reasonable number of parallel machines that could be available to a testing organization.

RQ2 - Parallel Speedup: Does the performance of PRSS improve with the number of parallel machines used? If so, is there a point of diminishing returns?

RQ3 - Fault Detection: Can PRSS be used to detect a failure in programs where the default exploration times out or runs out of memory?

RQ4 - Metrics Correlation: Do machine-independent metrics for state-space exploration correlate with real time for exploration? Does the correlation differ on compute clusters and dedicated machines?

Subject	Source	Error	#Threads	#Classes	SLOC
Airline	[24]	Assertion violation	6	2	136
BoundedBuffer	[24]	Deadlock	9	5	110
BubbleSort	[24]	Assertion violation	4	3	89
Daisy	[24]	Assertion violation	3	21	744
Deadlock	[24]	Deadlock	3	4	52
DEOS	[24]	Assertion violation	4	24	838
Elevator	[24]	ArrayIdxOOBExcptn	4	12	934
PoolOne	[1]	Assertion violation	3	51	10042
PoolTwo	[2]	Assertion violation	3	35	4473
PoolThree	[3]	Deadlock	2	51	10802
RaxExtended	[24]	Assertion violation	6	11	166
ReplicatedWorkers	[24]	Deadlock	3	14	432
RWNoDeadLckCk	[24]	Assertion violation	5	6	154

Table 3.1: Study Artifacts

RQ5 - Metrics Selection: Which metrics should be reported in future studies on state-space exploration?

3.2 Study Setup

Artifacts: We conducted our study with the thirteen concurrent Java programs shown in Table 3.1. The programs have diverse characteristics and include benchmarks obtained from the Software-artifact Infrastructure Repository (SIR) [9, 24] and real-world test cases obtained from the Apache Commons Pool project [1–3].

We performed our JPF-based experiments with the eleven programs shown in Table 3.2, which includes all seven programs used in the original PRSS study with the same inputs. The only exception is BoundedBuffer for which the latest JPF runs out of 8GB of memory for all explorations with the (3,6,6,1) input used in the original PRSS study. Since an older version of JPF was able to detect an error that the latest version could not detect, we have reported this as a regression to JPF developers, who have confirmed the bug and are investigating it. Therefore, in our experiments we used the default input of (1,4,4,2). Airline and Deadlock were not used for the JPF experiments since the default exploration found the failure too quickly to warrant the use of PRSS.

We performed our ReEx based experiments with the seven programs shown in Table 3.3 and Table 3.4. The remaining six programs are reactive programs with cyclic state spaces that cannot be explored with ReEx (or another stateless tool that performs no state matching or fair scheduling to make progress when cycles are possible), because any single execution of such a program could be infinitely long.

Experiments: We conducted six sets of experiments to answer our research questions:

- **JPF-Cluster**: We performed default depth-first exploration and 1000 randomized depth-first explorations using the latest JPF for each of the eleven programs used for the JPF-based experiments. We used the `DFSHeuristic` search class with the `cg.seed` property and the `cg.randomize_choices` property set to `path` to perform the randomized depth-first explorations with the latest JPF. Using the results of these experiments we were able to *revisit PRSS with a different version of JPF* and reconsider the PRSS-related research questions. The real time measurements from these experiments were used to partially answer the research questions related to correlation of real time with machine-independent metrics.

- **ReEx-DFS-Cluster**: We performed default depth-first exploration and 500 randomized depth-first explorations using ReEx for each of the seven programs used for the ReEx-based experiments. We used the `RandomDepthFirst` search strategy with the `reex.exploration.randomseed` property to perform the random depth-first explorations with ReEx. Using the results of these experiments we were able to *revisit PRSS with a stateless exploration tool*. The real time measurements from these experiments were also used to partially answer the research questions related to real time.

- **ReEx-ICB-Cluster**: We performed default ICB exploration and 500 randomized ICB explorations using ReEx for each of the seven programs used for the ReEx-based experiments. The default ICB exploration was performed with the `IterativeContextBounding` search strategy and the `reex.exploration.preemptionbound` property set to `2`. The

random ICB explorations were performed using the `RandomIterativeContextBounding` search strategy with the `reex.exploration.randomseed` property and the value 2 set for the `reex.exploration.preemptionbound` property. Using the results of these experiments we were able to *revisit PRSS with a different search strategy*. The real time measurements from these experiments were also used to partially answer the research questions related to real time.

- **Desktop**: We repeated a subset (50 seeds) of each of the cluster experiments on a dedicated desktop machine to make more accurate real time measurements and contrast them with real time measurements from the cluster experiments.

The cluster experiments were performed on our departmental compute cluster with 375 machines with CPU configurations that were either an Intel Xeon CPU X5650 @ 2.67 GHz or Intel Xeon CPU L5420 @ 2.50 GHz, memory configurations ranging from 1GB to 8GB of ECC RAM, 10,000 RPM SAS hard disk drives, and 64-bit Sun JVM v1.6.0_10 on Linux 2.6.18. The desktop experiments were performed on a dedicated machine with an Intel Core2 Duo E8400 @ 3.0 GHz, 2.00GB RAM, a solid-state drive, 64-bit Sun JVM v1.6.0_29 on Microsoft Windows 7; each seed was re-explored for 3 samples, and the reported real times were averaged to account for the potential noise in the system. In total, across all the experiments, we performed 21775 explorations.

As in the original PRSS study, we ran 50 simulations for PRSS configurations with 1, 2, 5, 10, 15, 20, and 25 parallel computers—the larger number of computers was beyond the point of diminishing returns (PDR) [10]—using the results of each of the three cluster experiments in order to compute the effectiveness of PRSS for the new contexts. After computing the result distributions for the various configurations, we followed the same procedure as in the original PRSS study to arrive at the PDR value, with several researchers agreeing on a common value for the number of computers. To measure the correlation of various machine-independent metrics with real time, we built linear regression models of the various machine-

independent metrics vs. real time for each of the six experiments. The results of the experiments are presented in Section 3.4.

3.3 Study Design

Independent Variables: The independent variables for our study include the artifacts used, type of exploration tool used (stateful JPF or stateless ReEx), and type of search performed (depth-first or ICB). Another independent variable for the simulations performed to evaluate PRSS effectiveness is the number of parallel computers in the simulated configurations.

Dependent Variables: The measured results of our experiments in terms of both machine-independent and machine-dependent metrics form the basis for the dependent variables of our study. For RQ1 and RQ2, the dependent variable is the performance of the various PRSS configurations in terms of States for JPF and Schedules for ReEx and also the PDR in terms of the number of parallel computers. For RQ3, the dependent variable is the detection of a failure by PRSS configurations in cases where the default exploration was unable to do so. For RQ4 and RQ5, the dependent variable is the coefficient of determination (R^2) of the linear regression models built to fit the real time measurements to various machine-independent metrics.

3.4 Study Results

In this section we present the results of our experiments in terms of the research questions posed in Section 3.1. RQ1, RQ2, and RQ3 address the effectiveness of PRSS under new contexts. Figure 3.1 and Table 3.2 show PRSS results for the latest JPF version. Figure 3.1 shows the distribution of results for the various PRSS configurations. Table 3.2 shows the exploration costs for the default exploration, comparing it with the exploration costs for the

Subject	States			PDR		
	Default	Minimum	Maximum	Nodes	Mean	Speedup
BoundedBuffer	235	87	310	5	115	2.0
BubbleSort	258	71	768	5	173	1.5
Daisy	45712	19	44619	10	644	71.0
DEOS	28523	17075	470825	10	38019	*0.8
Elevator	468055	46424	4552786	5	100678	4.7
PoolOne	489	59	995	5	325	1.5
PoolTwo	603	46	1338	5	78	7.7
PoolThree	218	12	355	2	21	10.4
RaxExtended	4331	77	15111	5	139	31.2
ReplicatedWorkers	1960	109	29065	5	1853	1.1
RWNoDeadLckCk	396	62	12799	5	158	2.5

Table 3.2: JPF PRSS Results (* indicates a slowdown)

Subject	Schedules			PDR		
	Default	Min	Max	Nodes	Mean	Speedup
Airline	48	1	1191	2	1	48.0
BoundedBuffer	1691647 (TO)	1	1730643	10	583	$\geq$2901.6
BubbleSort	109	1	3016458	2	1	109
Deadlock	298	1	298	2	5	59.6
PoolOne	569914 (TO)	1	1118764	10	12	$\geq$47492.8
PoolTwo	596015 (TO)	1	795960	1	1	$\geq$596015.0
PoolThree	480381 (TO)	1	156921	5	1	$\geq$480381.0

Table 3.3: ReEx DFS PRSS Results

PRSS configuration indicated by the PDR. The table also shows the minimum and maximum exploration costs across all the 1000 random explorations performed for the experiment. Figure 3.2 and Table 3.3 show the PRSS results for stateless DFS exploration with ReEx. Figure 3.3 and Table 3.4 show PRSS results for ICB exploration performed with ReEx. RQ4 and RQ5 address the correlation of machine-independent metrics with real time. Table 3.5 shows results for the machine-independent metrics collected during the (stateful) JPF based experiments. The table presents R^2 values for the linear regression models built to fit real time to the machine-independent metrics. The models were built both for the experiments performed on the compute cluster and dedicated desktop machine. Table 3.6 shows the same results for machine-independent metrics collected during (stateless) ReEx based experiments.

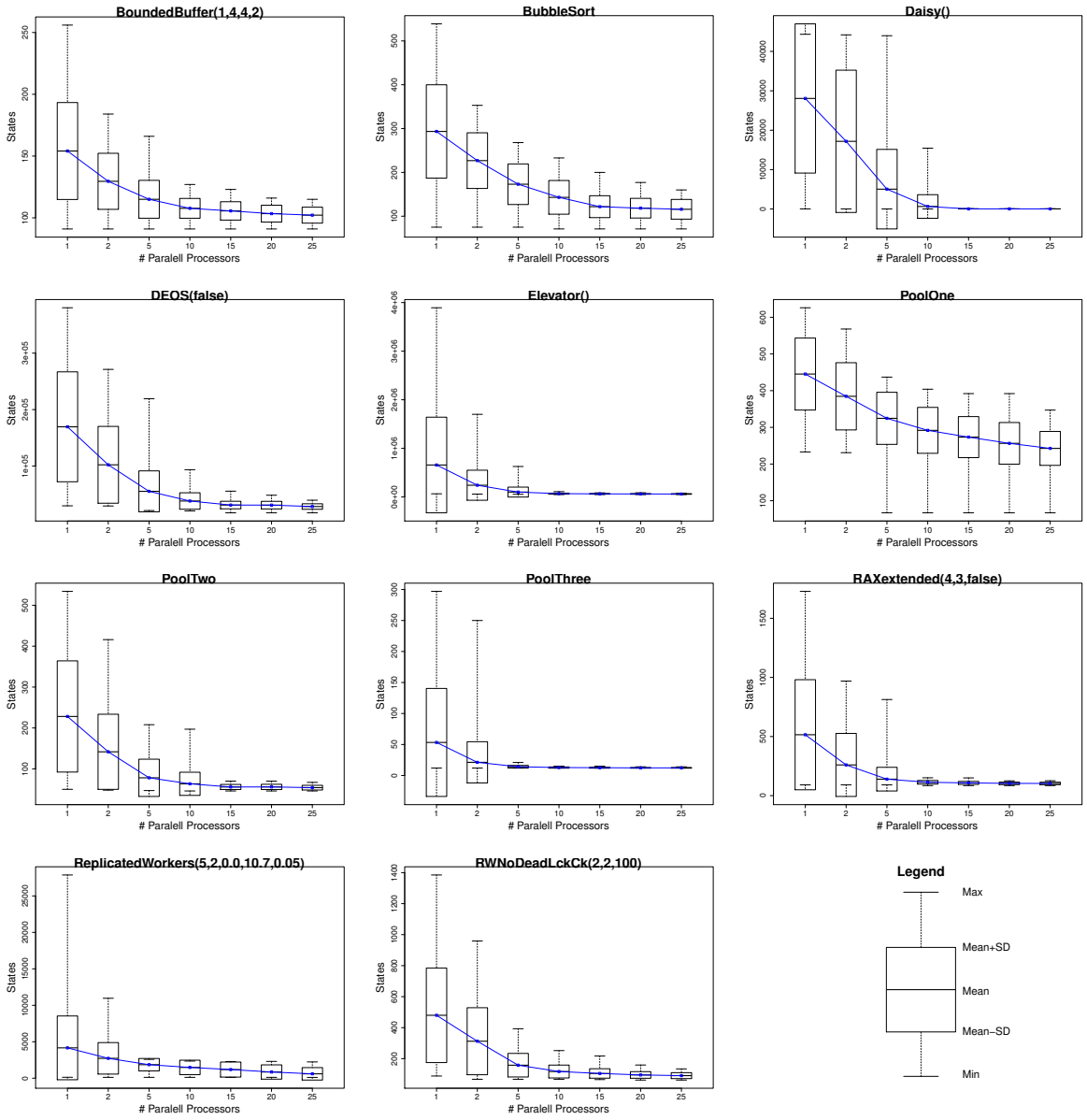

Figure 3.1: PRSS results for JPF

Subject	Schedules			PDR		
	Default	Min	Max	Nodes	Mean	Speedup
Airline	44312	42505	44448	5	44255	1.0
BoundedBuffer	1329	1	95	5	2	664.5
BubbleSort	5179	4644	4823	5	4648	1.1
Deadlock	6	3	16	5	3	2
PoolOne	6463	165	780	5	172	37.6
PoolTwo	123	3	71	10	4	30.7
PoolThree	2	1	2	1	1	2

Table 3.4: ReEx ICB PRSS Results

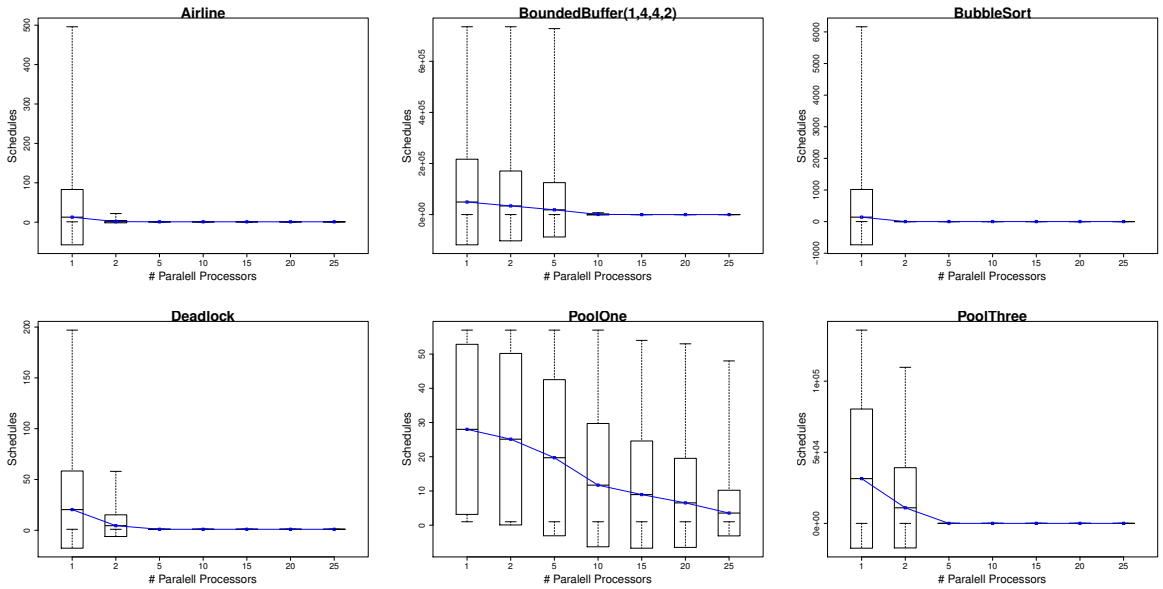

Figure 3.2: PRSS results for ReEx DFS

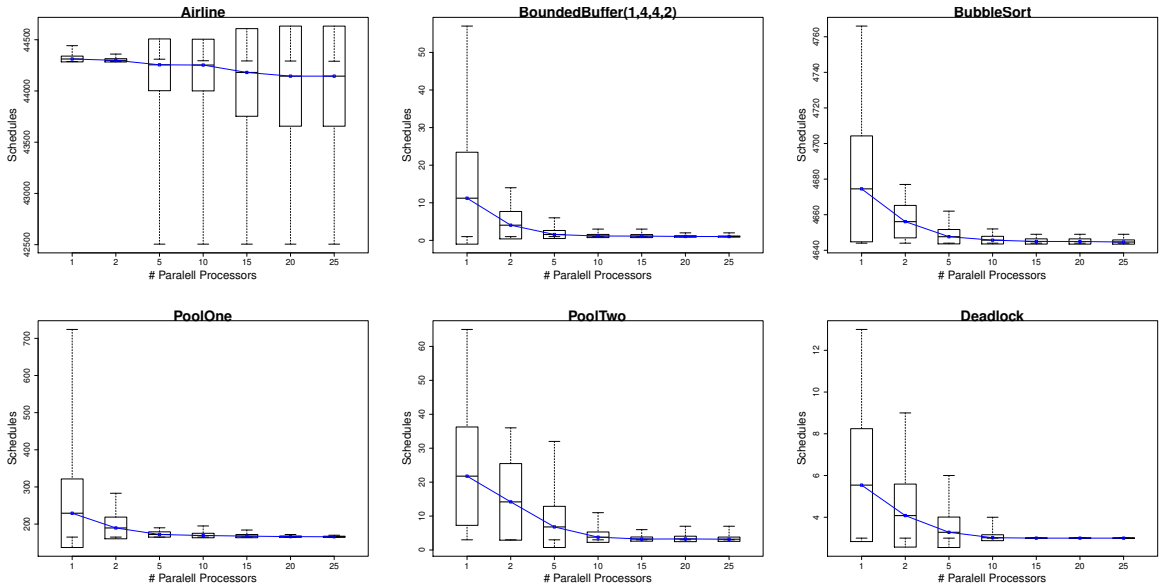

Figure 3.3: PRSS results for ReEx ICB

3.5 RQ1 - Cost Reduction

We discuss this question with respect to the new contexts under which we revisited PRSS.

Latest JPF (version 6.0): Comparing Table 3.2 with the original PRSS [10], the results indicate that the performance of JPF has improved substantially since the original PRSS

Subject	Desktop				Cluster			
	States	Trans	Insns	ThreadsCG	States	Trans	Insns	ThreadsCG
BoundedBuffer	0.943	0.935	0.887	0.908	0.039	0.041	0.045	0.040
BubbleSort	0.866	0.873	0.755	0.780	0.133	0.140	0.124	0.132
Daisy	0.995	0.995	0.999	0.995	0.849	0.849	0.849	0.849
DEOS	0.998	1.000	1.000	0.848	0.772	0.775	0.776	0.668
Elevator	0.999	0.999	0.998	0.999	0.921	0.921	0.921	0.921
PoolOne	0.581	0.566	0.511	0.570	0.072	0.069	0.065	0.072
PoolTwo	0.694	0.614	0.787	0.692	0.386	0.362	0.392	0.387
PoolThree	0.889	0.892	0.875	0.889	0.521	0.520	0.517	0.520
RaxExtended	0.972	0.948	0.746	0.961	0.756	0.714	0.439	0.734
ReplicatedWorkers	0.995	0.993	0.994	0.995	0.738	0.737	0.735	0.738
RWNoDeadLckCk	0.990	0.981	0.959	0.991	0.742	0.720	0.690	0.742

Table 3.5: JPF DFS Metrics Time Correlations

Subject	Desktop				Cluster			
	Schedules	Choices	Events	Threads	Schedules	Choices	Events	Threads
Airline	0.887	0.894	0.891	0.893	0.873	0.876	0.874	0.875
BoundedBuffer	1.000	0.998	0.999	1.000	0.908	0.910	0.911	0.908
BubbleSort	1.000	1.000	1.000	1.000	0.984	0.975	0.982	0.984
Deadlock	0.941	0.941	0.941	0.941	0.589	0.588	0.589	0.589
PoolOne	0.801	0.799	0.801	0.801	0.887	0.888	0.891	0.887
PoolTwo	0.000	0.018	0.000	0.000	0.943	0.945	0.947	0.943
PoolThree	0.884	0.886	0.884	0.884	0.972	0.958	0.972	0.972

Table 3.6: ReEx DFS Metrics Time Correlations

study. For most of the programs, the default exploration with the latest JPF finds the failure faster than reported in the original PRSS study. This is most evident with RaxExtended, ReplicatedWorkers, and RWNoDeadLckCk where the default exploration with the latest JPF is orders of magnitude faster than what was reported earlier. Despite this improvement in default JPF performance, all cases have a feasible PRSS configuration for every program, which could find a failure substantially faster than the default exploration (Table 3.2, Maximum and Minimum columns). For Daisy, all the PRSS configurations are able to find the failure faster, and for BoundedBuffer and PoolThree almost all the PRSS configurations are able to find the failure faster.

Stateless Exploration Tool: The results indicate that PRSS can be applied successfully even with a stateless exploration tool. For all programs there was a feasible PRSS configuration that substantially reduced exploration costs compared to the default exploration. For

Deadlock, all PRSS configurations were faster or as fast as the default exploration. For all programs, the fastest random search (Table 3.3, Minimum column) was substantially (orders of magnitude) faster than the default exploration. However, the slowest random searches (Table 3.3, maximum column) were also orders of magnitude slower than the default exploration (except for Deadlock).

ICB Search Strategy: The results for the ICB search strategy show that the effectiveness of PRSS is dependent on the search strategy that is used. While there existed a feasible PRSS configuration for each program that improved upon the default exploration, the diversity among different configurations was minimal compared to the JPF and ReEx random depth-first explorations. This can be observed from the difference between the Minimum and Maximum values in Table 3.4 and the means of the various PRSS configurations in Figure 3.3. For example, for Airline, the costs of the random explorations ranged from 42505 schedules to 44448 schedules, while the cost of the default exploration was 44312 schedules. This reduction in diversity is a factor of the nature of ICB exploration where schedules with lower number of preemptions are explored before schedules which higher number of preemptions. Hence the randomization only happens among choices with the same number of preemptions resulting in lesser diversity compared to random depth-first exploration.

Different Programs: There existed feasible PRSS configurations for all the additional programs evaluated that could provide speedup compared to their default explorations. This demonstrates that PRSS generalized across various types of programs.

3.6 RQ2 - Parallel Speedup

We discuss this question with respect to the new contexts under which we revisited PRSS.

Latest JPF (version 6.0): The improvement in the default performance for the latest JPF has resulted in a reduction in the magnitude of performance improvement random explorations can achieve, and also reduced the frequency of random explorations that perform

better than the default exploration. The effect of this was two-fold. While PRSS was able to achieve more than 100X speedup for three of the programs with the older JPF, the maximum speedup across all programs with the latest JPF was 71.0x. PRSS even resulted in a *slowdown* for exploring DEOS with the latest JPF, while it had obtained a 1.8x speedup with the older JPF. However, encouragingly, along with the reduction in speedup, the number of computers/nodes for the PDR PRSS configuration also reduced for the latest JPF. To summarize, *PRSS obtained lesser speedup with the latest JPF, but also required fewer parallel computers to achieve the reduced speedup.*

Stateless Exploration Tool: The PDR configurations for stateless exploration achieve significant speedups compared to the default exploration, the minimum speedup being 48.0X for Airline, and the maximum speedup being $\geq$596015.0X for PoolTwo. However, note that for PoolTwo, it is not really necessary to perform PRSS since the PDR configuration is the one with one parallel computer, i.e., any one random depth-first exploration is expected to find the failure so much faster than the default exploration. The next highest speedup after PoolTwo is $\geq$480381.0X for PoolThree, which is still many orders of magnitude. *While PRSS achieves much higher speedups for stateless exploration compared to stateful exploration, it does so with no increase in the number of parallel computers required. For all the programs, the PDR configuration required 10 or fewer parallel computers.*

ICB Search Strategy: As can be seen from Table 3.3 and Table 3.4, PRSS achieves much lesser speedup for the ICB search strategy compared to depth-first search. This can be attributed, as described earlier, to the lack of diversity in the randomized ICB explorations.

Different Programs: PRSS was able to achieve speedups for all the additional programs that we evaluated it with. The speedups ranged from 1.5X for BubbleSort and PoolOne on stateful JPF exploration to $\geq$480381.0X for PoolThree on stateless ReEx.

3.7 RQ3 - Fault Detection

Unlike in the original study, none of the default explorations timed out with the latest JPF. So we were unable to address this question with respect to those experiments. However, four of the seven programs used in the stateless ReEx experiments did time out for the default exploration (indicated by TO in Table 3.3). This included BoundedBuffer which was used in the original PRSS study and PoolOne, PoolTwo, and PoolThree, which were not used in the original study. For all four of these programs, PRSS was able to find the failure and achieve significant speedup even compared to the costs at the point of timeout. There were no default explorations that timed out for the ICB experiments so this question could not be addressed for that context.

3.8 RQ4 - Metrics Correlation

Stateful JPF: The Desktop section of Table 3.5 shows that for almost all the programs, all the machine-independent stateful exploration metrics considered correlate highly with real time, with R^2 values generally more than 0.85. The only exceptions are PoolOne and PoolTwo for which all the metrics have lower R^2 values, around 0.6. It is interesting to note that all the considered metrics correlate equally well with real time, i.e., *metrics that measure exploration costs with a finer granularity (e.g., instructions), which could result in a higher measurement overhead, do not provide benefits in terms of higher correlation with real time.*

The Cluster section of Table 3.5 shows that the metrics considered do not correlate with real time as well as in the Desktop experiments. However, surprisingly, for the majority of programs, all the stateful exploration metrics do correlate reasonably well with real time, with R^2 values generally more than 0.75. Also, as observed in the Desktop experiments, all the metrics considered correlate equally well (or equally badly) with real time.

Stateless ReEx: Table 3.6 shows that for almost all the programs, all the machine-independent stateless exploration metrics considered correlate highly with real time. This is the case for both the Desktop experiments and *surprisingly* the Cluster experiments as well. The only exceptions are PoolTwo for the Desktop experiments and Deadlock for the Cluster experiments.

ICB ReEx: Due to the lack of diversity in the random ICB explorations and their fast completion times, we were unable to build meaningful linear regression models comparing the real time measurements from these experiments with machine-independent metrics measurements. Hence, we do not consider these results in our discussions.

3.9 RQ5 - Metrics Selection

The experimental results show that all the commonly considered machine-independent metrics correlate equally well (or equally badly) with real time. Thus, researchers can continue to use the metrics that are currently used in the literature. In fact, *counter intuitively*, metrics that measure exploration cost more accurately—like instructions for stateful exploration and events for stateless exploration—do not provide substantial benefit in terms of correlation with real time.

It is commonly acknowledged in the research community that real time measurements from experiments performed on compute clusters may not be reliable measures. Hence when experiments are performed on compute clusters, the authors usually do not report the real time measurements. However, *surprisingly*, our results indicate that real time measurements from clusters can be useful since they correlate well with machine-independent metrics. So in the future, researchers should report real time measurements from cluster experiments.

3.10 Threats to Validity

Internal Threats: We performed our experiments with the default settings in JPF and ReEx, with a time bound of one hour. Changing the settings or increasing the time bound could affect our findings. To the best of our knowledge, there are no bugs in both tools and our implementations of randomized depth-first search and randomized ICB search algorithms in ReEx that affected our results. However, we did encounter a regression error in JPF that we reported to the JPF developers.

External Threats: We used a diverse set of thirteen artifacts in our experiments as shown in Table 3.1. The artifacts include benchmark programs obtained from SIR [9,24], which have been used in many previous studies. The artifacts also include real-world test cases obtained from the Apache Commons Pool project [1–3]. However, these programs do not necessarily form a representative set of concurrent programs. Also, the programs that we used exhibit a diverse set of failures including deadlocks, atomicity violations, and data races that lead to various assertion violations. However, these failures may not form a representative set of failures found in all concurrent programs.

To mitigate the effects of using a single type of exploration tool or a single search strategy, we used both stateful JPF and stateless ReEx for both DFS and ICB search in our experiments. However, this does not cover all types of exploration tools or search strategies that may be used, hence there remains a threat that our results may not generalize to other types of tools or search strategies, or combinations of search strategies [13].

Our experiments were performed across two different types of machine configurations, including a desktop machine and a heterogeneous cluster, using two different operating systems and two different JVM versions. However, these machine configurations may not necessarily form a representative set of all machine configurations. More combinations of both desktop and cluster machines with configurations including different operating system parameters, such as disk swapping, and different JVMs, possibly with alternate parameters

such as heap size, would be needed to eliminate this threat.

Construct Threats: While designing our study, we chose metrics that are used most commonly to report results for stateful and stateless state-space exploration experiments. However, using different metrics or new combinations of them may shed a different light on our results.

Conclusion Threats: We used 1000 random seeds for the JPF based experiments performed on the cluster, 500 random seeds for both the ReEx based experiments performed on the cluster and a subset of 50 of those seeds for the three experiments performed on the dedicated desktop machine. While these are a reasonably large number of seeds, there exists a threat that they may not have sufficiently represented the actual distribution of all possible random explorations.

While our cluster experiments were performed at different times of the day across the span of a few weeks, the real time results obtained could be a consequence of the load patterns prevalent at those times. Controlled experiments with different load patterns would have to be performed to rule out this threat.

Chapter 4

Conclusions and Future Work

State-space exploration of concurrent code is an increasingly important testing problem that is being actively researched. Many new techniques are proposed and evaluated for reducing the costs of state-space exploration. Empirical evaluations of these techniques use various machine-independent and machine-dependent metrics, which can make it hard to compare techniques.

Our results show that several machine-independent metrics for state-space exploration correlate well with real time, the machine-dependent metric of primary importance to the end users. Based on these results we provide a guideline that researchers can use for selecting and reporting metrics in the future studies on state-space exploration. We also revisit and extend the PRSS study [10], and our results indicate that PRSS is still useful for the latest JPF and also useful for stateless search in ReEx.

Our study considered five research questions in four new contexts across two different types of machine configurations, but there is much that can be done to build upon our research. Future work could extend our experiments in several different ways by considering the following: different search strategies and settings for both JPF and ReEx, different state-space exploration tools, new artifacts exhibiting different concurrency bugs, new types of metrics, new linear regression models that consider combinations of different types of metrics, more random seeds, different machine configurations, and different load patterns for clusters. All of these extensions would add to the diversity and scale of data already presented in this thesis, which could lead to new conclusions or eliminate threats.

References

[1] Apache Software Foundation, *POOL-107*, `https://issues.apache.org/jira/browse/POOL-107`.

[2] ———, *POOL-120*, `https://issues.apache.org/jira/browse/POOL-120`.

[3] ———, *POOL-146*, `https://issues.apache.org/jira/browse/POOL-146`.

[4] *ASM Library home page*, `http://asm.ow2.org/`.

[5] Richard H. Carver and Yu Lei, *Distributed reachability testing of concurrent programs*, Concurrency and Computation: Practice and Experience **22** (2010), no. 18, 2445–2466.

[6] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin, *Parallel symbolic state-space exploration is difficult, but what is the alternative?*, Parallel and Distributed Methods in Verification, 2009, pp. 1–17.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled, *Model checking*, The MIT Press, Cambridge, MA, 1999.

[8] Katherine Coons, Sebastian Burckhardt, and Madanlal Musuvathi, *Gambit: Effective Unit Testing for Concurrency Libraries*, PPoPP '10: The 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2010.

[9] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel, *Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact*, Empirical Software Engineering **10** (2005), no. 4, 405–435.

[10] Matthew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Rahul Purandare, *Parallel randomized state-space search*, Proceedings of the 29th International Conference on Software Engineering (ICSE'2007), 2007, pp. 3–12.

[11] Matthew B. Dwyer, Suzette Person, and Sebastian G. Elbaum, *Controlling factors in evaluating path-sensitive error detection techniques*, Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'2006), 2006, pp. 92–104.

[12] Milos Gligoric, Vilas Jagannath, and Darko Marinov, *MuTMuT: Efficient exploration for mutation testing of multithreaded code*, Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'2010), 2010, pp. 55–64.

[13] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce, *Tackling large verification problems with the swarm tool*, SPIN Workshop on Model Checking Software, 2008, pp. 134–143.

[14] Vilas Jagannath, Qingzhou Luo, and Darko Marinov, *Change-aware preemption prioritization*, International Symposium on Software Testing and Analysis (ISSTA 2011), 2011.

[15] *JPF home page*, `http://babelfish.arc.nasa.gov/trac/jpf/`.

[16] Madanlal Musuvathi and Shaz Qadeer, *Iterative context bounding for systematic testing of multithreaded programs*, PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, 2007, pp. 446–455.

[17] Madanlal Musuvathi and Shaz Qadeer, *Fair stateless model checking*, Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08, 2008, pp. 362–371.

[18] Pavel Parizek and Ondrej Lhoták, *Randomized backtracking in state space traversal*, SPIN Workshop on Model Checking Software, 2011, pp. 75–89.

[19] Soyeon Park, Shan Lu, and Yuanyuan Zhou, *CTrigger: Exposing atomicity violation bugs from their hiding places*, ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, 2009, pp. 25–36.

[20] *ReEx home page*, `http://mir.cs.illinois.edu/reex/`.

[21] Neha Rungta, Eric G. Mercer, and Willem Visser, *Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution*, Proceedings of the 16th International SPIN Workshop on Model Checking Software, 2009, pp. 174–191.

[22] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan, *Penelope: weaving threads to expose atomicity violations*, ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 37–46.

[23] Abhishek Udupa, Ankush Desai, and Sriram Rajamani, *Depth bounded explicit-state model checking*, SPIN Workshop on Model Checking Software, 2011, pp. 57–74.

[24] University of Nebraska Lincoln, *Software-artifact Infrastructure Repository*, `http://sir.unl.edu/portal/index.html`.

[25] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda, *Model checking programs*, Automated Software Engineering **10** (2003), no. 2, 203–232.

[26] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel, *Regression model checking*, IEEE Internation Conference on Software Maintenance, 2009, pp. 115–124.