

Parallel Randomized State-space Search*

Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, Rahul Purandare
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{dwyer,elbaum,sperson,rpuranda}@cse.unl.edu

Abstract

Model checkers search the space of possible program behaviors to detect errors and to demonstrate their absence. Despite major advances in reduction and optimization techniques, state-space search can still become cost-prohibitive as program size and complexity increase. In this paper, we present a technique for dramatically improving the cost-effectiveness of state-space search techniques for error detection using parallelism. Our approach can be composed with all of the reduction and optimization techniques we are aware of to amplify their benefits. It was developed based on insights gained from performing a large empirical study of the cost-effectiveness of randomization techniques in state-space analysis. We explain those insights and our technique, and then show through a focused empirical study that our technique speeds up analysis by factors ranging from 2 to over 1000 as compared to traditional modes of state-space search, and does so with relatively small numbers of parallel processors.

1. Introduction

The first general tool for model checking programs [12] was developed nearly ten years ago. The realization that variants of temporal logic model checking algorithms could be applied to search the space of possible program behaviors, to detect errors and demonstrate their absence, has spurred a tremendous body of research in the past decade. Much of this work has been oriented towards developing general techniques for reducing the analysis cost through property preserving state-space reductions, e.g., [16, 5], and abstraction techniques, e.g., [1, 13]. Another line of research has adapted model checking algorithms and data structures to optimize error detection while sacrificing the ability to demonstrate the absence of errors. Notable suc-

cess has been achieved along these lines for sequential program analysis, for example, analysis and detection of classes of errors in the linux kernel [22], TCP/IP implementations [19], and widely-used file system implementations [29].

Success in detecting concurrency-related errors on software of realistic scale and complexity, however, has been more difficult to achieve. The key complicating factor with concurrency is the need to analyze program behavior under the set of possible schedules that could be produced by the run-time system. In general, the set of all possible executions grows exponentially with the number of threads of control in a program. Concurrency errors, such as deadlocks and data-inconsistencies that arise due to data-races, can be very difficult to detect since they may only be exhibited on a small fraction of the possible program executions.

Systematic search of a program's feasible state-space, i.e., the set of control and data configurations that can be reached along some program execution, is attractive for these *hard to find* errors, since, given sufficient time and memory the error will eventually be revealed. Unfortunately, even when the full-complement of state-of-the-art state-space reduction techniques are applied, there are programs for which such an analysis will exhaust available time and/or memory before detecting the error [6]. In this paper, we address the challenge of providing additional reductions in analysis cost by *exploiting knowledge we have acquired studying program state-space structure as it relates to error states, and using this knowledge to create a technique that parallelizes the analysis.*

Our insight on parallelization opportunities emerged from our recent investigation of how the order in which a state-space is searched influences the cost and effectiveness of detecting errors [6]. Our empirical study of 56 multi-threaded Java programs showed that random variations in the search order give rise to enormous variations in the cost to find an error across a space. It was common, for example, to find programs where, given a few hundred random searches, the fastest search order outperformed the slowest by four or five orders of magnitude.

*This work was supported in part by the National Science Foundation through awards 0429149, 0444167, 0541263 and through CAREER award 0347518, by the Army Research Office through DURIP award W911NF-04-1-0104, and by NASA JPL through contract 1286178.

Ideally, to improve the efficiency of the error detection process, one would like to guide the model-checker towards regions of the program state-space that contain errors, and avoid regions that are free of errors. Distinguishing such regions without first exploring them, however, is beyond the current state-of-the-art in search heuristics. Instead, we have developed a technique, which we call Parallel Randomized State-space Search (PRSS), that runs multiple parallel randomized state-space searches, and terminates all searches when the first one finds an error. The intuition behind PRSS is that by sampling different regions of the state-space, there is a good chance that a region containing errors will be found. In addition, by exploring regions in parallel, the time required to search regions that do not have errors is mitigated. Our evaluation of the PRSS technique on the most challenging of the multi-threaded Java programs from our previous study demonstrates that PRSS can reduce the cost to find an error using state-space search by factors ranging from 2 to well over 1000, and that this reduction can be achieved using a relatively small number of parallel processors, ranging from 5 to 20.

In addition to improving the cost to find an error, PRSS has a number of other benefits. For example, PRSS is a general technique that can be composed with existing reduction, abstraction and heuristic techniques to further enhance the gains achieved by those techniques. Furthermore, it appears to be broadly applicable across a range of programs. Its performance benefits accrue when run on numbers of processors that most developers will have ready access to, for example, in a handful of multi-core workstations. In principle, PRSS could be implemented using any explicit state model checker or similar state-space analysis tool. In this paper we report on results using version 3.1.2 of Java PathFinder [27].

The contributions of this paper lie in (i) the presentation of a practical and cost-effective technique for detecting hard to find errors in concurrent programs, which we detail in the next Section, and (ii) the results of an empirical study that provide evidence of the effectiveness of the PRSS technique instantiated for Java PathFinder as compared to using the default mode of analysis with Java PathFinder over a range of non-trivial multi-threaded Java programs. Section 4 describes our study design and setup, and we present and discuss the results of the studies in Section 5. We discuss related work in Section 6 and describe plans for further assessing the effectiveness of PRSS in Section 7.

2. Motivation for PRSS

In previous work [6], we discovered that randomizing the order of program state-space search can *sometimes* lead a model-checker to locate an error state very quickly, outperforming a model checker’s default search order. This is not very surprising. Given enough randomized searches one is

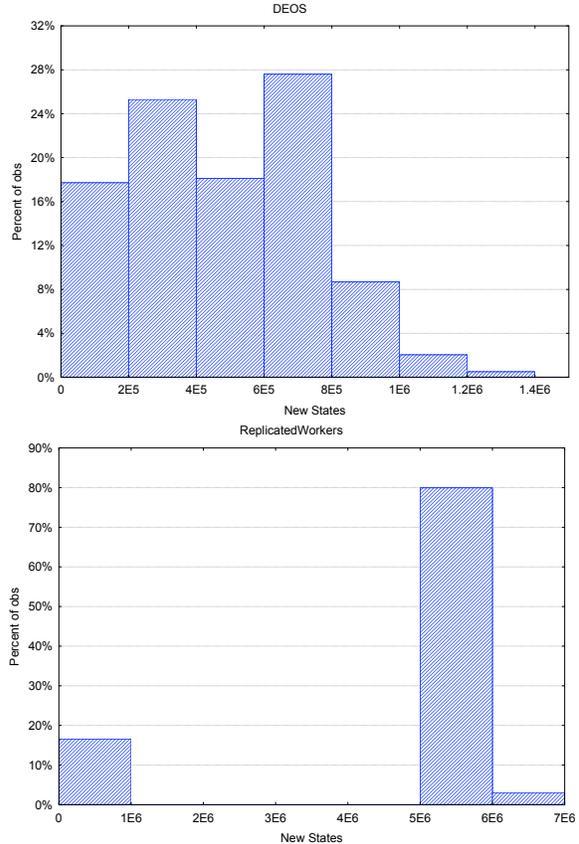


Figure 1. Search Cost Distributions

bound to find a search that detects errors more quickly than the default search order, which is generally defined without regard to program structure or the type of error.

What was surprising however, was the degree of variation in the cost of search across different programs. Some programs exhibited cost distributions that were flat, indicating that searches of varying cost were equally likely, some were clustered, indicating that all searches within a given group had similar cost, some were close to Gaussian, and some were bipolar, i.e., two clusters at the low and high-end of the cost scale. Figure 1 illustrates cost distributions for two of the programs in our study utilizing histograms. The x-axis represent the number of states visited by the model-checker and each bar represents the percentage of 5000 randomized depth-first search runs performed on the given program. With DEOS and ReplicatedWorkers we observed variations in cost that spanned one or more orders of magnitude; this is representative of the population of programs we studied. The key observation we made was that despite this enormous range, there were always some relatively low-cost runs, on the order of 10s or 100s of thousands of visited states that detected the error. For example, DEOS and ReplicatedWorkers have 18% and 17% of their runs that found the error in this low-cost region, re-

```

DFS(s)
4  workSet := enabled(s)
5  for each  $\alpha \in \textit{workSet}$  do
6    s' :=  $\alpha$ (s)
7    if error(s') then
basicDFS()
1  seen := {s0}
2  push(stack, s0)
3  DFS(s0)
end basicDFS()
8    counterexample := stack
9    exit
10   if s'  $\notin$  seen then
11     seen := seen  $\cup$  {s'}
12     push(stack, s')
13     DFS(s')
14     pop(stack)
end DFS()

```

Figure 2. Basic DFS for first error state

```

randDFS(seed)
1  seen := {s0}
2  init_rand(seed)
3  push(stack, s0)
4  DFS(s0)
end randDFS()
SHUFFLE(seq)
5  for each i := 0 . . . |seq|
6    r := i + (rand() * (|seq| - i))
7    t := seq[r]
8    seq[r] := seq[i]
9    seq[i] := t
end SHUFFLE()
DFS(s)
10 workSet := SHUFFLE(enabled(s))
11 for each  $\alpha \in \textit{workSet}$  do
12   s' :=  $\alpha$ (s)
13   if error(s') then
14     counterexample := (stack, seed)
15     exit
16   if s'  $\notin$  seen then
17     seen := seen  $\cup$  {s'}
18     push(stack, s')
19     DFS(s')
20     pop(stack)
end DFS()

```

Figure 3. Randomized DFS

spectively.

This trend holds up across all of the 56 programs we studied in [6]; we also found at least some runs that were significantly less expensive than the mean cost across the set of randomized runs we explored. From this observation, we conjectured that *by performing enough randomized search runs we would eventually be able to find a run that can find an error quickly*. We leverage this conjecture by running the searches in parallel to reduce the wall-clock time for detecting errors, which led to our Parallel Randomized State-space Search technique.

3. The PRSS Technique

The PRSS algorithm is an integration of classic depth-first search (DFS) to find error states, randomized DFS, and parallelized search. We explain each of these aspects in turn by highlighting portions of the overall algorithm.

```

PRSS(N, seed)
1  init_rand(seed)
2  for each i := 1 . . . N
3    start(randDFS(rand()), i)
4  while (true)
5    for each j := 1 . . . N
6      if (done(j)) then
7        first := j
8        break while@4
9      endif
10   for each k := 1 . . . N
11     if (k  $\neq$  first) then
12       stop(k)
13   print(counterexamplefirst)
end PRSS()

```

Figure 4. Parallel randomized DFS

3.1. Depth-first State-space Search

Our analysis involves a stateful search of a program’s state-space. Researchers have proposed the use of stateless search, e.g., [24], but our experience using such searches indicated that it is not cost-effective for programs with hard to find bugs, i.e., where the percentage of executions of the program that exhibit the error is near zero. For example, on the `Elevator` program in our study, in over 3 hours of run-time, 10,000 randomized stateless searches were unable to detect the error, whereas our randomized stateful searches always found the error with a mean run-time of 6 minutes. We used depth-first search (DFS) as the basis for PRSS in this paper; we plan to explore the use of variants of breadth-first search in future work.

Abstractly we view a program as a guarded-transition systems and analyze transition sequences. A *guarded transition system* consists of a set of variables, which for our purposes are coalesced into a single composite *state variable* *s*, and a set of guarded transitions which atomically test, with predicate ϕ , the current state and update the state by executing a transition, α , i.e., if $\phi(s)$ then $s = \alpha(s)$. The initial values of program variables are used to define an initial state, *s*₀.

Figure 2 presents the basic DFS algorithm that generates the program state-space terminating when it finds an error or finds all reachable states. `basicDFS` initializes the set of states *seen* in the search, and the *stack* that stores the current path in the state-space being analyzed, and then starts a recursive chain of DFSs from the initial state. Lines 4-14 comprise a step in the DFS search. On line 4, *enabled*(*s*) returns the set of transitions, α , whose guard, ϕ , is true in the given state. Line 5 iterates through the set of enabled transitions and we assume that the order of iteration is fixed, i.e., it is the same for every every run of the algorithm, which is the default for all existing state-space analysis tools that

we are aware of. Lines 7-9 test if an error state has been reached, and if so, record the current DFS stack, which encodes the path under analysis, as a counterexample and exits.

3.2. Randomized State-space Search

Researchers in randomized testing [2] have explored the use of randomized sampling of a program’s input domain to detect errors. In contrast, we randomize the sequence of scheduling decisions that are made by the underlying runtime system in executing a program.

Randomization of DFS is achieved by applying a Fisher-Yates shuffle [17], lines 5-9 of Figure 3, to the sequence of enabled transitions at each state explored. Each time the algorithm executes, the order in which enabled transitions is explored on line 10 is randomized. This approach to randomization has the advantage that reduction techniques that operate by modifying the set of enabled transitions, such as partial-order reductions for Java [5], can be applied first and then the sequence in which the remaining transitions are explored is randomized. Randomization in the shuffle follows a pseudo-random sequence whose seed is passed as a parameter to `randDFS`, in Figure 3, and used to initialize the sequence on line 2. When an error is detected the analysis returns the seed along with the sequence of program transitions as a counter-example (line 14). This allows replay of randomized runs to analyze counter-examples in detail.

3.3. Parallel State-space Search

PRSS, shown in Figure 4, accepts a parameter (N) that controls the degree of parallelism to be applied in the analysis and a parameter (`seed`) that gives users control over the randomization in the algorithm; passing the same seed provides reproducibility whereas passing a random sequence of seeds provides effective randomization. The analysis starts N copies of a randomized DFS (lines 1-3) each with a different seed that is calculated based on a pseudo-random sequence that is initialized with the seed parameter.

There are many different implementation strategies that can be applied to distribute jobs to nodes in a parallel machine or distributed cluster. We describe a polling approach based on three abstract primitives: `start(m, i)` executes method m on machine i , `done(i)` polls to determine if the job on machine i is complete, and `stop(i)` terminates the job on machine i . It would be a simple matter to map the logic of lines 4-9 to primitives that block until job completion rather than use this polling approach.

When a job completes it will be detected within N calls to `done` and its index is then recorded as the *first* to complete (line 7) and the polling loop is exited (line 8); we are not concerned with the minor differences in run-time that would

arise due to races among jobs completing at approximately the same time. Lines 10-12 shutdown all other executing jobs and the counterexample from the *first* job is printed.

There are several notable aspects of this algorithm. (1) Unlike many existing approaches to parallelization of state-space search, which we discuss in detail in Section 6, PRSS is *embarrassingly parallel* [9]. The N parallel randomized depth-first searches are performed completely independently such that state information collected and used by each search job is kept local to the job and need not to be exposed in any way to the other parallel searches. This eliminates the need for costly inter-process communication and coordination between jobs.

(2) PRSS runs multiple simultaneous state-space searches in distinct portions of the state-space; the likelihood of two searches ending up in the same region of the state-space is low. By using multiple randomized searches to explore a single state-space, the *chance* that one search will explore a region that is relatively dense with error states is increased over a single search, and the *penalty* for searching in a region that is free of errors is mitigated since a sibling search may be making progress at the same time.

(3) PRSS leverages all of the optimizations applied to the underlying DFS algorithms and its precision is limited only by the precision of the underlying DFS. Note that it neither creates additional behavior nor removes existing behavior in the state-space and therefore does not affect the soundness of the underlying search technique.

4. Study

The purpose of our study was to evaluate the cost and effectiveness of PRSS for error detection. We set the study in the context of a collection of Java programs containing concurrency-related defects, and compared the performance and fault detection capabilities of PRSS at various degrees of parallelism against JPF’s default search settings. We used JPF’s `RandomOrderScheduler` to implement the `randDFS` algorithm in Figure 4. The specific PRSS configurations evaluated, specified as the number of parallel randomized searches, are described in Section 4.1.1. For this study we investigated the following research questions:

RQ1: (Cost Reduction) Does there exist a feasible configuration of PRSS that can detect a program error more quickly than performing a state-space search using the default search order? Where, by *feasible*, we mean a number of parallel processing nodes that might reasonably be available to a software testing organization.

RQ2: (Parallel Speedup) Does the performance of PRSS improve with increased parallelism? If so, is there a point of diminishing return?

Subject	Source	Parameters	Error	# Threads	Classes	SLOC
BoundedBuffer(3,6,6,1)	[4]	modCount, bufferSize, #producers, #consumers	Deadlock	13	5	65
Daisy()	[21]	none	AssertionViolation	3	21	744
DEOS(false)	[10]	abstracted?	AssertionViolation	4	24	838
Elevator()	[7]	none	ArrayIdxOOBExcpn	4	12	934
RaxExtended(4,3,false)	[10]	gc, wc, envFirst?	AssertionViolation	6	11	127
ReplicatedWorkers(5,2,0.0, 10.7,0.05)	[4]	#workers, #items, min, max, epsilon	Deadlock	6	14	304
RWNoDeadLckCk(2,2,100)	[4]	#readers, #writers, bound	AssertionViolation	5	6	103

Table 1. Study artifacts

RQ3: (Fault Detection) Can PRSS be used to detect an error in programs where the default searcher fails because of insufficient time or space?

4.1. Characterization Variables

4.1.1 Independent Variable

To answer our research questions, we manipulated one independent variable: the number of parallel randomized state-space searches. For practical purposes, this measure represents the number of *parallel processors* or *nodes* used when applying the PRSS technique. Because there is no fixed upper bound on the number of parallel searches one might perform, and because it would be impractical for a study such as this to attempt to test every potential node configuration, we chose 11 different configurations including 1, 2, 5, 10, 15, 20, 25, 50, 100, 500, and 1000 parallel nodes. Our goal was to select a set of practical values that includes a sufficient number and range of data points to be able to identify trends in cost and performance.

4.1.2 Dependent Variables

The dependent variable for RQ1 and RQ2 is *tool performance*. We measure performance in terms of the number of program states explored. We use this measure because it is platform-independent and it is a common metric for evaluating state-space exploration tool performance, such as model checker performance. In JPF, this metric is referred to as the number of *new states*.

For RQ3, the dependent variable is fault detection capability. This variable is simply a measure of whether the technique detects the program fault or not. Each technique is tested under the same conditions (i.e. resource constraints) which means that the opportunity to detect the program error is equal for all techniques.

4.2 Artifacts

Seven unique concurrent Java programs form the collection of artifacts for our study. All programs exhibit a single concurrency error represented as a deadlock, an exception, or an assertion violation. Table 1 describes the programs.

The programs were selected from the population of 56 parameterized artifacts used in [6]. Because this study is focused on *hard to find* defects, we limited the selection of artifacts to all but one of the programs that were classified as "realistic." This class of programs contains Java artifacts that perform a computation over rich data structures, many of which have been previously used in slightly different forms to evaluate Java state-space search techniques in the literature. The only "realistic" program from that study that was not used is *AlarmClock*. This particular program was omitted from the current study because, although it is interesting in some contexts, its small state-space does not challenge state-of-the-art search techniques.

4.3. Study Design and Setup

To conduct this study, we needed to evaluate the artifacts on each of the parallel search configurations. This required a minimum of 1,728 randomized searches per artifact, i.e., the sum of the configuration sizes mentioned in Section 4.1.1 per artifact.

Based on our previous experience, where we observed that program state-spaces can be extremely large and that the number of states visited before detecting the program defect can vary greatly, we chose to evaluate each artifact 50 times for each parallel search configuration. This meant we required 86,400 searches per artifact, and 604,800 searches total for seven artifacts.

To control the costs of the conducting the study, we chose instead to produce a pool of 5000 random searches for each artifact, from which n searches would be randomly selected to represent a configuration of n parallel searches for each experiment. The pool size of 5000 was selected based on our previous experience.

The following steps were then performed to obtain our study results. For each program artifact:

1. We performed 5000 random searches using JPF version 3.1.2 on a cluster of dual-Opteron 250's running at 2.4 GHz with 16GB of memory and running Fedora Core 3 Linux. Each randomized search used a distinct seed generated from a pseudo-random sequence, and was limited to one hour of execution time and 2GB of memory, with the exception of `BoundedBuffer`. Higher bounds (14GB and four hours) were used for `BoundedBuffer` in order to evaluate the PRSS technique on a program with a larger state-space.
2. To simulate a run of n parallel randomized searches for a given artifact, we randomly sampled, with replacement, the pool of 5000 randomized searches for that artifact n times. We repeated this sampling process to produce a total of 50 *trials* to account for potential variation across samples.
3. From each sample of size n , we chose the search with the shortest time to represent the search that would have completed first if the searches had actually been performed in parallel. In the case of a tie, one search result was selected from the group.

4.4 Threats to Validity

In this section, we describe the internal, external, construct and conclusion threats to the validity of this study. We also include the approaches we designed to minimize the impact of these threats on our findings.

Internal threats. Setting different bounds for the model checker can clearly impact the findings. For example, unlimited time and memory would allow all searches to find the program defect. Conversely, for some searches, one might expect that increasing the time or memory bound might simply allow the analysis to *take longer* to exhaust those resources. Our choice for upper bound on time and memory for JPF was primarily meant to be consistent with settings used in other recent studies.

External threats. Our study was performed on a single state-space search tool - JPF version 3.1.2. Different versions of JPF or different state-space analysis tools may yield different results. Replicated studies with different versions of JPF or with different tools would address this threat. The artifacts chosen for this study may also affect the results. We selected artifacts classified as "realistic" programs from the population of artifacts used in [6] in an attempt to evaluate the effectiveness of PRSS on detecting hard to find defects. We do not know, however, if these artifacts and the defects they contain are truly representative of hard to

find defects in the broader population of multi-threaded Java programs.

Construct threats. The measures we selected for this study provide what we believe are a reasonable way to evaluate its results. However, other measures may provide perspectives that we did not consider. Nevertheless, to be consistent with other studies and more relevant to the model checking community, we decided to use the number of new states which is platform-independent and commonly used in evaluating model checking and other state-space analysis tools.

Conclusion threats. In order to execute this study, we chose to simulate each parallel, randomized search for a given artifact by randomly selecting a search from the pool of 5000 randomized searches performed on that artifact. It is possible that the pool size of 5000 randomized searches per artifact is not sufficiently diverse to accurately represent the set of all feasible randomized searches for that artifact. It is also possible that the number of trials (50) performed on each artifact for each parallel randomized search configuration does not accurately represent the set of feasible results. We attempted to mitigate these threats by choosing the pool size and number of trials based on the experiences gained in our previous study. For example, in our previous study, 500 randomized searches produced a stable variance in the number of states visited to first error for some of our artifacts but not all. We therefore set the pool size at 5000, an order of magnitude larger, in an attempt to achieve a more stable variance in all artifacts. Overall, given the exploratory nature of the study at this point we do not consider limited pool size to be a major source of concern.

5. Study Results

Figure 5 provides a graphical depiction of the results of our study in a series of seven plots, one per program. Within each plot, for each PRSS configuration, we show the mean cost, in new states explored, and the standard deviation in cost over the 50 trials we evaluated. We only show data up to 25 parallel nodes for PRSS for all of the programs except `BoundedBuffer`, where we show data up through 50 parallel nodes. Only these configurations were included in the graphs because the trends are nearly flat and unchanging beyond these points.

The plots include three additional reference lines. *Default States* and *Min States* represent the *Default* and *Minimum* values from Table 2, respectively. Note that in some cases the default value is off the scale and therefore not shown, because the default search either ran out of memory or exceeded the time limit. The *100% Runs Completed* line indicates the point at which all of the 50 trials for the PRSS configurations to the right of the line completed and found the error; to the left of this line, at least one of the 50

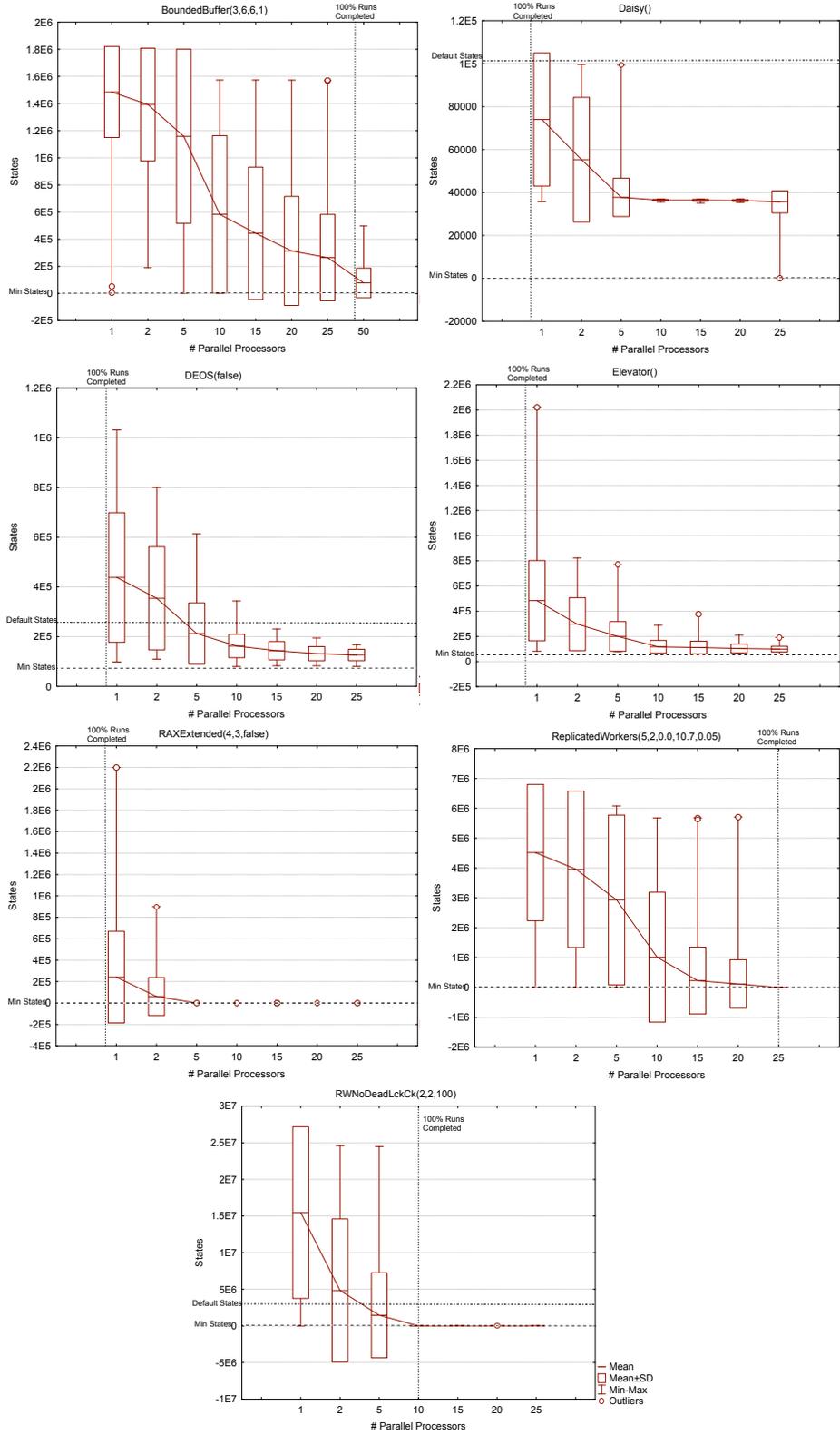


Figure 5. Scaled PRSS performance

Artifact	States				PDR		
	Default	Found w/ Default	Minimum	Maximum	Nodes	Mean States	Speedup
BoundedBuffer(3,6,6,1)	2603200	OM	1702	1573469	20	313290	≥ 8.3
Daisy()	101816	✓	140	99865	5	37715	2.7
DEOS(false)	260039	✓	75919	1465215	15	143860	1.8
Elevator()	11743698	TO	57082	6751854	10	116960	≥ 100.4
RaxExtended(4,3,false)	3470398	TO	41	3176159	5	176	≥ 19718.1
ReplicatedWorkers(5,2,0.0,10.7,0.05)	6231840	TO	372	6642260	15	226790	≥ 27.5
RWNoDeadLckCk(2,2,100)	3147356	✓	40	24796751	10	1847	1704.0

Table 2. Results Summary

trials of parallel searches at a given PRSS node configuration either ran out of memory or exceeded the time bound.

In Table 2 we summarize the results of our study. We show the number of states explored by the default search and indicate if the search completed (✓), timed out (TO) or ran out of memory (OM). The *Minimum* and *Maximum* values are the *observed* minimum and maximum number of states explored by the random searches in the pool. The Point of Diminishing Returns (PDR) values are explained in Section 5.2. In the remainder of this section, we consider each of the research questions in turn.

5.1. RQ1 - Cost Reduction

The plots in Figure 5 clearly indicate that, for our study, there is always at least one, and often many, feasible PRSS configurations capable of detecting an error more quickly than the default search. In the case where the default search does not complete execution (i.e., times out or runs out of memory), this observation still holds because the number of states explored by the default search, as presented in Table 2, can be viewed as an under-approximation of the actual number of states that would need to be explored in order to detect the error.

For `Elevator` and `RaxExtended`, even running a single randomized DFS finds the error in all of the trials we performed with one node. This is remarkable given the size of the state space searched by the default run before running out of resources.

For `Daisy` and `DEOS`, simply performing a single randomized DFS may not yield a more efficient analysis according to our experiment; however, increasing the parallelism to 2 and 15 nodes, respectively, for these examples beats the default in all 50 trials.

`RWNoDeacLckCk` shows a similar trend, but with the additional fact that below 10 parallel randomized DFSs there is a possibility that one or more randomized searches fails to complete - even when the default completes. At 10 nodes, however, PRSS beats the performance of default by a factor of 1700, has almost no variation in this performance across the 50 trials, and never fails to find

the error in our experiment. `ReplicatedWorkers` and `BoundedBuffer` show similar trends where a degree of parallelism of 25 and 50, respectively, is needed to achieve 100% error detection according to our study. Based on these findings, it seems clear that *there exist feasible configurations of PRSS that can detect a program error more quickly than performing a state-space search using the default search order.*

5.2. RQ2 - Parallel Speedup

The plots in Figure 5 share a characteristic shape. For all artifacts, the curve has a downward trend as parallelism is increased and a *leveling off* towards higher degrees of parallelism. These plots confirm that *the performance of PRSS improves with increased parallelism.* Furthermore, as parallelism increases, the variation in performance observed decreases. This is because a larger degree of parallelism effectively increases the sample size of the set of randomized searches and the likelihood of finding an *inexpensive* search increases.

By inspecting these plots, we are able to approximate a *Point of Diminishing Returns* (PDR) which is an estimate of the degree of parallelism beyond which additional computational resources provide increased performance that is not justified by those extra resources. Our definition of PDR is informal and intuitive: all of the authors of this paper studied the data and determined what they believed the PDR to be. We agreed on the PDR for all but one example, `DEOS`, where some authors thought the value was 10 and others thought 15.

Table 2 shows the relatively small number of parallel nodes corresponding to the PDR; in all cases, we found this number to be less than 20. The table also shows the speedup of the PDR configuration of PRSS over the default search; speedups for artifacts whose search did not finish are considered lower bounds. These indicate the benefits of using PRSS. The variation in speedups is enormous, but all of the examples exhibit non-trivial speedup and many have an order of magnitude or more speedup. For some examples, it is clear that there are more efficient searches

that could be performed with more nodes than the number we identified as the PDR. For example, `BoundedBuffer` and `ReplicatedWorkers`, speedups of 33.8 at 50 nodes and 36231.6 at 25 nodes, respectively are achieved.

5.3. RQ3 - Fault Detection

In choosing the artifacts for this study, our goal was to choose programs that contain *hard to find* defects. Of the seven artifacts selected, four have defects that were not detected by using the default search order because they either timed out or ran out of memory. For all of those artifacts, we were able to use PRSS to consistently find the error given a sufficient level of parallelism.

For `Elevator` and `RaxExtended`, all configurations of PRSS found the defect in our experiments in all of the 50 trials performed. For `ReplicatedWorkers` and `BoundedBuffer`, the error was consistently detected when the degree of parallelism was increased to 25 and 50 nodes, respectively. We conclude that *PRSS can be used to detect an error in artifacts where the default searcher fails because of insufficient system resources.*

Since finding errors in large multi-threaded programs is not cost-effective using existing state-space search approaches, some researchers have turned to more modular approaches where an application is broken into pieces and those pieces are analyzed independently [25, 26]. It would be interesting to explore the application of PRSS to those applications to see if errors can be detected without the added costs associated with modular reasoning, for example, through the construction of environments that simulate the calling context of a program component.

6. Related Work

Given the computational cost of state-space search, it is natural to wonder whether it can be effectively parallelized. Stern and Dill report on the parallelization of the `Mur ϕ` model checker [23]. Their approach stands as the model upon which all other techniques in the literature are based. They distribute a collection of searches targeting portions of the state-space rooted at different nodes. A shared *seen* set is used to keep searches from performing redundant work. This set must be locked to ensure coherent updates. The overhead of locking, and the poor locality in the sub-state-spaces searched in parallel, cause this algorithm to scale poorly. Researchers have explored the use of lock free shared structures, to minimize contention, and dynamic load balancing [18], but even with those improvements the coordination of multiple searches seems to greatly limit scalability. Our approach is embarrassingly parallel, so it has no coordination overhead, but it may do ar-

bitrary amounts of redundant work, which reduces the useful parallel work it performs.

The idea of using randomization in state-space search dates back to West [28] who showed that it can be effective in finding bugs in large protocols. It is supported in modern tools, for example, JPF has had the `RandomOrderScheduler` component for several years, but it's combination with parallel execution had not yet been explored or validated empirically until our work.

Randomization in state space search can be used to control the schedulings explored, as in our work, or to control which states are stored in the *seen* set. To control memory requirements, techniques like bit-state hashing [15] randomly drop states from the *seen* set. While lossy, this approach can scale analyses to very large problems. In [14], multiple bit-state hashing runs are explored in parallel to reduce the time to find errors. This approach is very similar to ours except that our approach is not lossy, since the underlying randomization technique is not lossy. While [14] describes the techniques use for large systems, there is no empirical study of the performance improvements seen when parallelization and randomization are used together in different configurations.

Recent work has developed the concept of Monte Carlo Model Checking [11] which computes a bound on the probability that randomized walks of the state-space, beyond a specified value, will find an error; this is not a bound on the probability that an error exists. We make no attempt to estimate the benefits of additional randomization, but instead observe empirically that relatively small numbers of samples seem sufficient for error detection.

Randomization in software testing is an old idea [2] that has proven to be effective in practice [8]. Approaches for randomizing the input space of a program under constraints designed to improve error-detection have been proposed [3, 20] and they seem to be effective. These techniques do not target concurrent executions explicitly and make no attempt to randomize the scheduler's behavior. This may make them less effective at revealing concurrency errors. It would be interesting to adapt the intuition of these approaches to randomized scheduling [24]. Our approach considers program input to be fixed, and rather than performing a stateless search, we randomize a stateful search. Our experience suggests that both the randomness and the state-fullness are key ingredients to its success.

7. Conclusions and Future Work

We have presented a simple and cost-effective technique for amplifying the benefits of existing optimizations for state-space search targeted at error detection. We believe this approach to be broadly compatible with explicit-state model checking approaches and applicable across a wide

range of programs. Further empirical studies would be valuable in validating this belief, but the results from our study suggest that a practical and significant cost-reduction can be achieved in the analysis of programs with large state-spaces.

In the future, we would like to explore the impact of parallelization and randomization on other forms of state-space search, such as variants of breadth-first and heuristic searches. Heuristics tend to focus a search on portions of the state space, but when the heuristic scoring function is discrete, multiple enabled transitions can receive the same score. Given that randomization appears effective in improving the performance of state-space search over the default order, it may also prove effective in shuffling the order in which *ties* are broken and thereby speed error detection for heuristic searches as well.

References

- [1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.
- [2] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.
- [3] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(101):1025–1050, Sept. 2004.
- [4] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of program slicing for model reduction of concurrent object-oriented programs. In *Proc. of the Twelfth Int'l. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, 2006. LNCS 3920.
- [5] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 25(2–3):199–240, September–November 2004.
- [6] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of the ACM SIGSOFT Fourteenth Int'l. Symp. on Foundations of Software Engineering*, 2006. to appear.
- [7] <http://home.att.net/ddavies/NewSmulator.html>.
- [8] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.
- [9] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [10] A. Groce and W. Visser. Heuristics for model checking java programs. *Int'l. Journal on Software Tools for Tech. Transfer*, 6(4):260–276, 2004.
- [11] R. Grosu and S. A. Smolka. Monte carlo model checking. In *Proceeding of 11th International Conference Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–286, 2005.
- [12] K. Havelund. Java pathfinder, a translator from java to promela. In *Theoretical and Practical Aspects of SPIN Model Checking*, page 152, 1999.
- [13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.
- [14] G. Holzmann and M. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, Apr. 2000.
- [15] G. J. Holzmann. An analysis of bitstate hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.
- [16] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *Software Tools for Technology Transfer*, 6(4):302–319, 2004.
- [17] D. E. Knuth. *The art of computer programming, volume 2 : seminumerical algorithms*. Addison-Wesley, 1997.
- [18] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification*, pages 19–34, Apr. 2005.
- [19] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. of the First Symp. on Networked Systems Design and Implementation*, Mar. 2004.
- [20] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.
- [21] <http://research.microsoft.com/qadeer/cav-issta.htm>.
- [22] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22. IEEE Computer Society, 2005.
- [23] U. Stern and D. L. Dill. Parallelizing the mur ϕ verifier. *Form. Methods Syst. Des.*, 18(2):117–129, 2001.
- [24] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Proc. Workshop on Runtime Verification*, 2002.
- [25] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th International Conference on Automated Software Engineering*, Oct. 2003.
- [26] O. Tkachuk and S. Rajan. Application of automated environment generation to commercial software. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA '06)*, July 2006.
- [27] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.
- [28] C. H. West. Protocol validation by random state exploration. In *Proceedings of the 7th Workshop on Protocol Specification, Testing and Verification*, 1986.
- [29] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proc. of the Seventh Symp. on Operating Systems Design and Implementation*, Dec. 2004.